

ASG-Manager Products™ Procedures Language

Version: 2.5.1

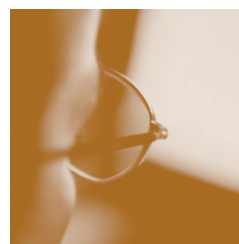
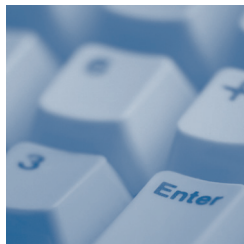
Publication Number: MPR0500-251-PROCL

Publication Date: November 2001

The information contained herein is the confidential and proprietary information of Allen Systems Group, Inc. Unauthorized use of this information and disclosure to third parties is expressly prohibited. This technical publication may not be reproduced in whole or in part, by any means, without the express written consent of Allen Systems Group, Inc.

© 1998-2001 Allen Systems Group, Inc. All rights reserved.

All names and products contained herein are the trademarks or registered trademarks of their respective holders.



ASG Worldwide Headquarters Naples, Florida USA | asg.com

1333 Third Avenue South, Naples, Florida 34102 USA Tel: 941.435.2200 Fax: 941.263.3692 Toll Free: 1.800.932.5536

ASG Documentation/Product Enhancement Fax Form

Please FAX comments regarding ASG products and/or documentation to (941) 263-3692.

Company Name	Telephone Number	Site ID	Contact name

Product Name/Publication	Version #	Publication Date
Product:		
Publication:		
Tape VOLSER:		

Enhancement Request:

ASG Support Numbers

ASG provides support throughout the world to resolve questions or problems regarding installation, operation, or use of our products. We provide all levels of support during normal business hours and emergency support during non-business hours. To expedite response time, please follow these procedures.

Please have this information ready:

- Product name, version number, and release number
- List of any fixes currently applied
- Any alphanumeric error codes or messages written precisely or displayed
- A description of the specific steps that immediately preceded the problem
- The severity code (ASG Support uses an escalated severity system to prioritize service to our clients. The severity codes and their meanings are listed below.)
- Verify whether you received an ASG Service Pack for this product. It may include information to help you resolve questions regarding installation of this ASG product. The Service Pack instructions are in a text file on the distribution media included with the Service Pack.

If You Receive a Voice Mail Message:

- 1 Follow the instructions to report a production-down or critical problem.
- 2 Leave a detailed message including your name and phone number. A Support representative will be paged and will return your call as soon as possible.
- 3 Please have the information described above ready for when you are contacted by the Support representative.

Severity Codes and Expected Support Response Times

Severity	Meaning	Expected Support Response Time
1	Production down, critical situation	Within 30 minutes
2	Major component of product disabled	Within 2 hours
3	Problem with the product, but customer has work-around solution	Within 4 hours
4	"How-to" questions and enhancement requests	Within 4 hours

ASG provides software products that run in a number of third-party vendor environments. Support for all non-ASG products is the responsibility of the respective vendor. In the event a vendor discontinues support for a hardware and/or software product, ASG cannot be held responsible for problems arising from the use of that unsupported version.

Business Hours Support

Your Location	Phone	Fax	E-mail
United States and Canada	800.354.3578 1.941.435.2201 Secondary Numbers: 800.227.7774 800.525.7775	941.263.2883	support@asg.com
Australia	61.2.9460.0411	61.2.9460.0280	support.au@asg.com
England	44.1727.736305	44.1727.812018	support.uk@asg.com
France	33.141.028590	33.141.028589	support.fr@asg.com
Germany	49.89.45716.300	49.89.45716.400	support.de@asg.com
Singapore	65.224.3080	65.224.8516	support.sg@asg.com
All other countries:	1.941.435.2201		support@asg.com

Non-Business Hours - Emergency Support

Your Location	Phone	Your Location	Phone
United States and Canada	800.354.3578 1.941.435.2201 Secondary Numbers: 800.227.7774 800.525.7775 Fax: 941.263.2883		
Asia	011.65.224.3080	Japan/Telecom	0041.800.9932.5536
Australia	0011.800.9932.5536	New Zealand	00.800.9932.5536
Denmark	00.800.9932.5536	South Korea	001.800.9932.5536
France	00.800.9932.5536	Sweden/Telia	009.800.9932.5536
Germany	00.800.9932.5536	Switzerland	00.800.9932.5536
Hong Kong	001.800.9932.5536	Thailand	001.800.9932.5536
Ireland	00.800.9932.5536	United Kingdom	00.800.9932.5536
Israel/Bezeq	014.800.9932.5536		
Japan/IDC	0061.800.9932.5536	All other countries	1.941.435.2201

ASG Web Site

Visit <http://www.asg.com>, ASG's World Wide Web site.

Submit all product and documentation suggestions to ASG's product management team at <http://www.asg.com/products/suggestions.asp>

If you do not have access to the web, FAX your suggestions to product management at (941) 263-3692. Please include your name, company, work phone, e-mail ID, and the name of the ASG product you are using. For documentation suggestions include the publication number located on the publication's front cover.

Contents

Preface	xi
About this Publication	xi
Publication Conventions	xii
1 Introduction	1
User-defined Commands	2
2 Basic Concepts	5
Setting Up Executive Routines	6
Using Executive Routines	6
From the Command Area	6
Line Command	6
Cursor Spatial Command	7
Components Of Executive Routines	7
Directives	8
Labels	8
Comments	9
Variables	10
Execution Control	10
Functions	12
User-defined Functions	13
COBOL Argument Block	13
PL/I Argument Block	14
Assembler Argument Block	15
General Notes on Argument Blocks	15
Outputting Information	16
Manipulating Buffers	17
Debugging	17

Parsing	18
Efficiency And Readability.....	20
3 Setting Up And Using Executive Routines	21
Introduction.....	21
Corporate Executive Routines.....	22
Setting Up EXECUTIVE-ROUTINE Members.....	23
Executing Executive Routines From The Command Area.....	24
Line Commands	24
Line Commands Example	25
Cursor Spatial Commands.....	26
Cursor Spatial Commands Example.....	27
Using Prefix Commands/Resolving Name Conflicts	27
4 Variables.....	29
Variables and Parameters	30
User-defined Variables.....	31
Arrays.....	31
System-assigned Variables	33
User-assigned Variables	33
Command Variables.....	33
Profile Variables	34
Global Variables	34
Local Variables	35
Parameter Variables.....	36
System Variables.....	37
&BUFN.....	37
&CCOD.....	37
&CCOL.....	37
&COLO.....	38
&CROW	38
&CURL.....	38
&CURS.....	38
&DATE.....	38
&DICT	38
&ECOD.....	39

&ENAM	39
&ENVO.....	39
&ENVM	39
&ENVT	40
&LINC	40
&LINO	40
&LOGO.....	40
&MODE	41
&MSLN.....	41
&MSLV.....	41
&MSNO	41
&MSTX.....	41
&PNUM	41
&PVAL.....	42
&SCOD.....	42
&STAT	42
&TIME	42
&TRMC	42
&TRMR	42
&USER.....	42
Return Codes.....	42
5 Expressions.....	45
Introduction.....	45
Full Evaluation	47
Numeric Expressions	49
6 Example Executive Routines	51
MP-AID Copy (PROCL-01).....	51
Condensed MP-AID List (PROCL-02).....	52
FASTQUIT (PROCL-03).....	54
Quick Sign On (PROCL-04)	54
Decimal Conversion (PROCL-05).....	55
All Occurrences (PROCL-06)	56
Overlay (PROCL-07)	57
PF Key Settings (PROCL-08)	58
Compound Interest (PROCL-09)	59
ISPF Read, Write and Edit	60

ISPF Variables.....	61
7 Executive Commands	63
ARRAYGEN	64
ARRAYGEN Syntax	65
ARRAYSORT	65
ARRAYSORT Syntax	66
BUILD	66
Building a KEPT-DATA List from an Array	67
Building an Array from a KEPT-DATA List	68
BUILD Syntax	69
CLOSEF Syntax	71
Obtaining Security, Current Status, and History Information	73
Obtaining Full Status Information	76
Suppressing Information	77
Maintaining Variables for Two or More Members	78
Obtaining Condition Information	80
Example	81
DACCESS Syntax	83
Expanding a Member for a Particular Language	85
Using a Specific Form and Version of Any Processed Items	85
Generating Local Names as Variables	86
Giving Specified Alias Names	86
Example	87
Maintaining Variables for Two or More Members	89
DEXPAND Syntax	91
DRELEASE	91
Rules on Releasing Variables	92
DRELEASE Syntax	93
DRETRIEVE	94
Retrieving Repeating Clauses	95
Retrieving Used-By or Reference Information	96
Example 1	99
Example 2	99
Specifying Variable Names	100
Suppressing Information	101
Accessing DEXPANDED Information	102
Example	103
Retrieving Unique Key Identifiers	103
Maintaining Variables for Two or More Members	104
DRETRIEVE Syntax	105
RELINQUISH	106
RELENQUISH Syntax	107

RESERVE	107
RESERVE Syntax	108
SENDF	108
Sending Output to a USER-MEMBER	109
Sending Output to a Sequential Dataset	110
Sending Output to a Partitioned Dataset	112
SENDF Syntax	112
SREAD	113
SREAD Syntax	114
8 Directives	115
CALL	116
CALL label-name Option	117
CALL ARRAY Option	117
COMMAND	118
DO	119
DROP	121
EXIT	121
Example 1	122
Example 2	122
GLOBAL	122
GOTO	123
Examples	123
IF	123
Example 1	125
Example 2	125
Example 3	126
Example 4	126
INTERPRET	126
Example 1	127
Example 2	128
Example 3	128
ITERATE	128
Example	129
LEAVE	129
Example	129
LITERAL	130
LOCAL	131

MESSAGE	131
Example 1	132
Example 2	132
MPR	133
MPRE	134
MPXX	134
Example.	135
NOP	135
PARSE	135
Example 1	136
Example 2	137
PARSEOPTION	137
PROFILE	137
RELEASE	138
Erasing All Variables of a Particular Type	139
Erasing a Selection of Variables of a Particular Type	139
RELEASE Syntax	140
RETAIN	140
RETURN	140
SET	141
Examples.	141
SIGNAL	142
STACK	143
TRACE	144
Example.	145
TRANSFER	145
VLIST	146
Listing Particular Variables.	147
Listing All Variables of a Particular Type	147
Listing a Selection of Variables of a Particular Type	147
Listing Particular Variables.	147
Listing All Variables of a Particular Type	147
Listing a Selection of Variables of a Particular Type	147
Examples.	148
VLIST Syntax	149
WRITEF	149
Example 1	151
Example 2	151

9 Functions	153
ABBREV	155
Examples	155
ARG	156
Examples	156
ARRAYHI	157
Example	157
ARRAYLO	157
Example	157
BIN	158
Examples	158
CENTER	158
Examples	158
CLIENTI	158
Example	159
CLIENTN	159
Example	159
CLIENTU	159
Example	159
COPIES	160
Examples	160
DB2TYPE	160
DIVCAPT	160
Examples	161
DIVOBJN	161
Example	161
DIVOBJT	161
Example	161
EDDATE	161
Example	162
Example	163
EXTRACT	163
DSN Keyword	165
Primary Command Keyword	165
Examples	166
LCOFF Keyword	166
FDO	166
Example	167

GETSVRM	167
Examples	167
GETTOKEN	168
Example	168
GETUDSN	168
Example	168
HEX	169
Examples	169
INSERT	169
Examples	169
Examples	170
LEFT	170
Examples	170
LENGTH	170
LENGTH function	171
Implicit Length Function	171
LOWER	171
Examples	171
MAX	172
Example	172
MEMTYPE	172
Example	173
MIN	173
Example	173
MPRAID	173
MPRCMPW	174
MPRDDPW	174
MPRSU	174
MPRUCLS	174
MPRUDSN	175
NDATE	175
Example	175
Example	176
Examples	177
PACK	177
Examples	177
PARSABLE	178

POS	178
Examples.....	178
PTIME	178
Examples.....	179
REPSTR	179
Examples.....	179
Example.....	180
RIGHT	180
Examples.....	180
ROOT	180
Examples.....	181
SEARCH	181
Examples.....	181
SERVERN	182
Example.....	182
STIME	182
Example.....	182
STRIP	182
Examples.....	183
SUBSTR Function.....	184
Implicit Substring Function.....	184
SUBTASK	185
SUBTASK Function.....	185
SUBTENV	186
Example.....	187
TRANSLAT	187
Examples.....	187
TRUNCATE	187
Examples.....	188
TYPE	188
The TYPE Function.....	188
Implicit Type Function.....	189
UPPER	189
Examples.....	189
VALUE	190
Example 1.....	190
Example 2.....	190
WORD	191
Examples.....	191

WORDINDEX	191
Examples	191
WORDLEN	191
Examples	191
WORDS	192
Example	192
10 Debugging	193
Introduction	193
SET TRACE	194
Procedures Language Trace: Selecting Procedures	194
Procedures Language Trace: Selecting Variables	195
Procedures Language Trace: Information Available	196
Manager Products Trace: Information Available	197
Examples	197
Output Media	198
SET TRACE Syntax	198
QUERY TRACE	199
QUERY TRACE Syntax	200
Glossary	201
Index	205

Preface

This *ASG-Manager Products Procedures Language* publication describes how to write executive routines. It is assumed that you have some knowledge of programming. ASG-Manager Products (herein called Manager Products) is an integrated set of dictionary/repository-driven products developed by ASG for use on IBM System/370, 30xx and 4300 series, and plug compatible, machines.

Allen Systems Group, Inc. (ASG) provides professional support to resolve any questions or concerns regarding the installation or use of any ASG product. Telephone technical support is available around the world, 24 hours a day, 7 days a week.

ASG welcomes your comments, as a preferred or prospective customer, on this publication or on the Manager Products.

About this Publication

The *ASG-Manager Products Procedures Language* consists of these chapters:

- [Chapter 1, "Introduction,"](#) gives you an introductory overview of the procedures language, with references to related information in other publications.
- [Chapter 2, "Basic Concepts,"](#) gives the main features of the procedures language.
- [Chapter 3, "Setting Up And Using Executive Routines,"](#) describes types of executive routine and the different ways of calling them.
- [Chapter 4, "Variables,"](#) describes the different types of variables and their uses.
- [Chapter 5, "Expressions,"](#) describes the different types of expression and how they are evaluated.
- [Chapter 6, "Example Executive Routines,"](#) gives examples of how you can use executive routines.
- [Chapter 7, "Executive Commands,"](#) describes specifications of commands that can only be used in executive routines.

- [Chapter 8, "Directives,"](#) gives specifications of instructions that control the order in which instructions are executed, or perform input or output.
- [Chapter 9, "Functions,"](#) gives specifications of elements of expressions that manipulate strings or give information about the environment.
- [Chapter 10, "Debugging,"](#) is an outline of the facilities available for debugging, and gives details of commands available for this purpose.

Publication Conventions

Allen Systems Group, Inc. uses these conventions in technical publications:

Convention	Represents
ALL CAPITALS	Directory, path, file, dataset, member, database, program, command, and parameter names.
Initial Capitals on Each Word	Window, field, field group, check box, button, panel (or screen), option names, and names of keys. A plus sign (+) is inserted for key combinations (e.g., Alt+Tab).
<i>lowercase italic</i> <i>monospace</i>	Information that you provide according to your particular situation. For example, you would replace <i>filename</i> with the actual name of the file.
Monospace	Characters you must type exactly as they are shown. Code, JCL, file listings, or command/statement syntax. Also used for denoting brief examples in a paragraph.
Vertical Separator Bar () with underline	Options available with the default value underlined (e.g., Y <u>N</u>).

The following conventions apply to syntax diagrams that appear in this publication.

Diagrams are read from left to right along a continuous line (the "main path"). Keywords and variables appear on, above, or below the main path.

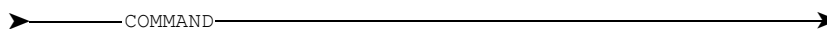
Convention	Represents
➤➤	At the beginning of a line indicates the start of a statement.
➤◀	At the end of a line indicates the end of a statement.
————→	At the end of a line indicates that the statement continues on the line below.
➤————	At the beginning of a line indicates that the statement continues from the line above.

Convention Represents

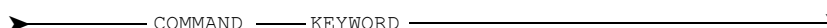
Keywords are in upper-case characters. Keywords and any required punctuation characters or symbols are highlighted. Permitted truncations are not indicated.

Variables are in lower-case characters.

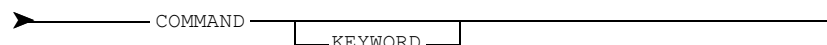
Statement identifiers appear on the main path of the diagram:



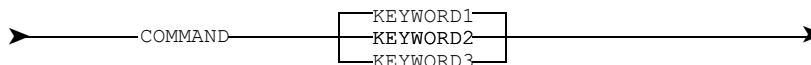
A required keyword appears on the main path:



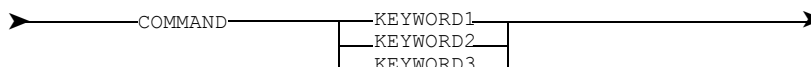
An optional keyword appears below the main path:



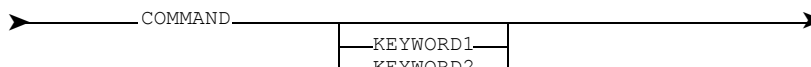
Where there is a choice of required keywords, the keywords appear in a vertical list; one of them is on the main path:



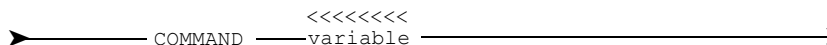
or



Where there is a choice of optional keywords, the keywords appear in a vertical list, below the main path:

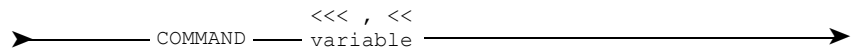


The repeat symbol, <<<<<<, above a keyword or variable, or above a whole clause, indicates that the keyword, variable, or clause may be specified more than once:

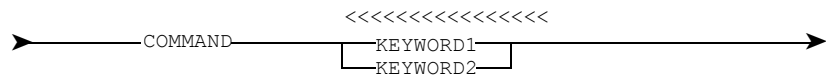


A repeat symbol broken by a comma indicates that if the keyword, variable, or clause is specified more than once, a comma must separate each instance of the keyword, variable, or clause:

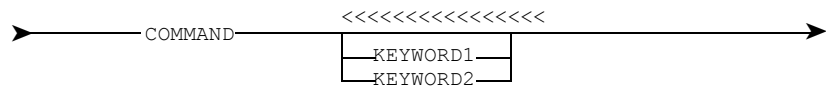
Convention	Represents
\mathbf{A}	Address
\mathbf{B}	Block
\mathbf{C}	Class
\mathbf{D}	Device
\mathbf{E}	Entity
\mathbf{F}	File
\mathbf{G}	Group
\mathbf{H}	Header
\mathbf{I}	Index
\mathbf{J}	Join
\mathbf{K}	Key
\mathbf{L}	Language
\mathbf{M}	Method
\mathbf{N}	Node
\mathbf{O}	Operator
\mathbf{P}	Process
\mathbf{Q}	Queue
\mathbf{R}	Record
\mathbf{S}	Schema
\mathbf{T}	Table
\mathbf{U}	User
\mathbf{V}	View
\mathbf{W}	Window
\mathbf{X}	Extension
\mathbf{Y}	Yield
\mathbf{Z}	Zone



The repeat symbol above a list of keywords (one of which appears on the main path) indicates that any one or more of the keywords may be specified; at least one must be specified:



The repeat symbol above a list of keywords (all of which are below the main path) indicates that any one or more of the keywords maybe specified, but they are all optional:



1

Introduction

The Procedure Language facility allows you to execute a sequence of instructions by storing them in executive routines. An executive routine is executed by entering its name.

These are the components of executive routines:

- Any Manager Products command normally enterable by the user (including calls to executive routines)
- Directives, which provide programming capabilities
- Function calls
- Labels
- Comments

All components of the procedures language may be entered as upper or lower case characters except for Manager Products commands, which must be entered in upper case.

Case is significant in strings. For example, these two strings:

```
abc  
ABC
```

are not equal.

The procedures language is free format, except for continuation lines, so you can indent instructions to make code more readable.

The systems administrator may set up executive routines (called corporate executive routines) which may then be executed by any user.

Users are able to set up their own executive routines (called user executive routines and transient executive routines).

Executive routines can take their parameters from:

- The Command Area
- A line on the screen (called a line command)
- The cursor position on the screen (called a cursor spatial command).

The systems administrator can rename any Manager Products command for use by general users so as to conform to local installation standards.

In executive routines PERFORM commands can be nested to any depth.

If your machine does not use the UK character set a few of your characters may be different from those printed in this manual. For example, the operator giving remainder after integer division (hex code 5A) is entered as square close bracket in the Belgian character set but entered as exclamation mark in the UK character set. It is printed as exclamation mark in this manual.

User-defined Commands

The User Defined Commands facility allows you:

- To use the procedures language
- To include primary commands in executive routines
- To use the following primary commands:
 - QUERY CORPORATE-EXECUTIVE-INDEX
 - SET and QUERY EXECUTIVE-RETENTION

With the Extended Interactive facility you may also:

- Set up transient executive routines
- Use the following commands:
 - MP-AID EXEC-LIST
 - SET and QUERY LINEAR-RETENTION
 - QUERY LINE-COMMANDS
 - SET and QUERY AUTOSKIP
 - SET and QUERY USER-DIRECTORY-SEARCH
 - QUERY USER-MEMBER-INDEX

The above commands are documented in the *ASG-ControlManager User's Guide*.

The User Defined Commands facility allows the systems administrator to use the following commands:

- SET PRIMARY-COMMAND
- SET CORPORATE-EXECUTIVE-INDEX

With the Extended Interactive facility, the systems administrator may also use the following commands:

- SET LINE-COMMAND
- SET USER-MEMBER-INDEX.

The above commands are documented in the *ASG-Manager Products Systems Administrator's Guide*.

These commands:

- DACCESS
- DEXPAND
- DRELEASE
- DRETRIEVE
- SENDF
- CLOSEF
- BUILD
- FORMAT
- TRANSFER
- TRANSLATE

are only available with the Translation and Transfer Engine facility.

If you have the Translation and Transfer Engine facility, but not the User Defined Commands facility then, apart from the above list of commands, you can only execute the following commands within executive routines:

- ADD
- REPLACE
- RESERVE
- RELINQUISH

2

Basic Concepts

This chapter includes these sections:

Setting Up Executive Routines	6
Using Executive Routines	6
From the Command Area	6
Line Command	6
Cursor Spatial Command	7
Components Of Executive Routines	7
Directives	8
Labels	8
Comments	9
Variables	10
Execution Control	10
Functions	12
User-defined Functions	13
COBOL Argument Block	13
PL/I Argument Block	14
Assembler Argument Block	15
General Notes on Argument Blocks	15
Outputting Information	16
Manipulating Buffers	17
Debugging	17
Parsing	18
Efficiency And Readability	20

Setting Up Executive Routines

Executive routines may be stored as EXECUTIVE members on the MP-AID by the system administrator. The contents of these members may subsequently be executed as corporate executive routines by general users.

Users can set up their own executive routines by storing them in USER-MEMBERS and TRANSIENTs and subsequently executing them as user executive routines or transient executive routines respectively.

Using Executive Routines

Executive routines can be used:

- From the Command Area
- As a line command
- As a cursor spatial command

From the Command Area

Enter the name of the executive routine in the Command Area followed by the values of any parameters.

For example, to execute an executive routine called SUBSTITUTE and to supply the values DISPLAY and 12 as the first and second parameters respectively, enter:

```
SUBSTITUTE DISPLAY 12
```

Line Command

Enter the name of the executive routine in the Line Command Area. The values of any parameters are derived from the contents of the associated data line so that the value of parameter &P0 is taken from the leftmost element of the line, the value of parameter &P1 is taken from the next element to the right, and so on.

Cursor Spatial Command

A parameter is supplied to the executive routine by reading a data element from the screen. There are two ways of making a data element available in an executive routine in system variable &CURS:

Method 1

- Assign a PF key to the name of the executive routine
- Place the cursor at the data element on the screen to be supplied as a parameter to the executive routine
- Press the PF key which has been assigned to the name of the executive routine.

Method 2

- Insert the name of the executive routine on the Command Line without pressing ENTER
- Move the cursor to the data element on the screen to be supplied as a parameter to the executive routine.

Components Of Executive Routines

Executive routines consist of the following components:

- Labels: these are reference points within an executive routine which may be branched to using the GOTO or SIGNAL directives.
- Comments: these are textual information intended to document the logic and/or usage of the executive routine.
- Instructions: these specify an action that Manager Products is to perform. They comprise all components of an executive routine except for labels and comments. Instructions may be any of the following:
 - Directives: these are instructions which either control the instruction sequence within an executive routine, or perform operations on data which is either internal to the executive routine or supplied by the user in the form of parameters. For example, IF is a directive.
 - Primary commands are commands which may operate on data which is either supplied by the user or is external to that generated by the executive routine. Primary commands may be issued outside executive routines in the Command Line or within executive routines. For example, LIST is a primary command.

- Executive commands are the same as primary commands in that they may operate on external data or parameters supplied by the user. However, they may not be executed outside executive routines as they are only applicable within the context of executive routines. For example, SENDF is an executive command.
- Executive routines may be called from within executive routines.

Directives

A directive is an instruction which either controls the instruction sequence within an executive routine, or performs operations on data which is either internal to the executive routine or supplied by the user in the form of parameters. For example, IF is a directive.

The following general points apply to directives:

- Directives may be in upper case, lower case or a mixture of the two
- Directives cannot be abbreviated.
- The procedures language is free format, so you are free to indent blocks of code in order to improve legibility. (This does not apply to continuation lines, see below.)
- Directives may be continued on one or more continuation lines by using the continuation character, a hyphen preceded by a space (' —'), as the last character on each line to be continued. The following line (the continuation line) is taken to start in column 1. The space and hyphen are replaced by the continuation line before the directive is executed.

Both space and hyphen are replaced by the continuation line. So if a line to be continued ends at the end of a word, the continuation line should start with a space.

Labels

A label is declared within an executive routine as follows:

```
-label
```

Labels consist of up to 50 alphanumeric characters and must be preceded by a hyphen. The hyphen is omitted in references to the label.

Excluding the initial hyphen, label identifiers must commence with a letter or a number. They must not contain imbedded blanks, but all other characters, including those normally regarded as literal and string delimiters, are permitted. Lower case and upper case alphabetic characters are treated as identical.

Before the executive routine is executed it is scanned for label declarations and label references to ensure that there are no label references that are undeclared or declared more than once.

There is no limit upon the number of labels that can be used in any one executive routine.

Unlike directives, label lines may not be continued on one or more following lines.

Labels are referenced from the CALL, GOTO and SIGNAL directives.

Comments

To make the whole of a line a comment, enter the two characters `/*` at the beginning of the line. For example:

```
/* The whole of this line is a comment.
```

To make the remainder of a line a comment, enter the characters `/*` at the point where you want the comment to start. For example:

```
SET OUTPUT-EDIT ON ; /* This is a comment.
```

To use `/*` or `/*` as literals enclose them in literal delimiters.

The following additional methods of commenting are retained merely for upwards compatibility. ASG recommends that you do not use them.

If the first non-blank character on a line is an asterisk (`*`), and this character is followed by a space, the remainder of that line is taken to be a comment. For example:

```
* This is a comment
```

Any text after the logical end of a directive or command is ignored and is effectively a comment. For example:

```
SET &LO ABC This is a comment.  
MPR LIST ITEMS ; This is a comment too.
```

Variables

A variable is a location to which a name is assigned and in which data can be stored within an executive routine. The value assigned to a variable may change during the execution of an executive routine. The value of a variable is a single text string of up to 255 characters which may contain any characters.

Variables are assigned values using the = operator. In the following example the value 123 is assigned to the variable *i*.

```
i = 123
```

There are several types of variables available with the procedures language. They can be divided into variables with user-defined variable names and variables with ASG-defined variable names.

Execution Control

Here is an example of a simple executive routine that takes three parameters and opens a repository.

```
MPXX LITERAL#  
DICTIONARY &P0 ;  
AUTHORITY &P1 ;  
STATUS &P2 ;  
SAY #Repository #&P0# now open for user #&P1# in status #&P2
```

The instructions are executed sequentially.

Instructions are executed sequentially unless you change the execution order using one of these directives:

- DO
- LEAVE
- ITERATE
- IF
- SIGNAL
- GOTO
- CALL
- RETURN.

DO is a loop construct. For example:

```
DO WHILE condition
  instruction_1
  instruction_2
  ...
instruction_n
END
```

The block is executed repeatedly while *condition* is true, that is, until *condition* becomes false. You can jump out of the block before *condition* becomes false using the LEAVE directive. You can skip an execution of the block using the ITERATE directive.

You can execute a block conditionally using:

```
IF condition THEN DO
  instruction_1
  instruction_2
  ...
instruction_n
END
```

The block is only executed if *condition* evaluates to true.

You can construct your own loop constructs using the IF and GOTO directives. For example:

```
-LABEL1                                /* repeat until
IF condition THEN GOTO LABEL2
instruction_1
instruction_2
...
instruction_n
GOTO LABEL1
-LABEL2
```

The block is executed repeatedly while *condition* is not true. You can call other executive routines. For example:

```
MPR MYEXEC
instruction
```

Control passes to the executive routine MYEXEC. When MYEXEC terminates, control passes back to the calling executive routine at the instruction *instruction*.

Functions

The procedures language provides a wide range of built-in functions. These include character manipulation, conversion and information functions. The user may also define external functions in another programming language (such as BAL, COBOL, and PLI).

All function calls, whether built-in or external, must be in the following format:

The diagram illustrates the syntax for a function call. It starts with a double right-pointing arrow followed by a horizontal line. This line leads to the text *function-name*, which is followed by an opening parenthesis *(*. After the parenthesis, another horizontal line leads to a box containing the text *<<< , <<<* above *argument*. This box is followed by a closing parenthesis *)*. Finally, a horizontal line leads to a double left-pointing arrow.

where *argument* is an expression which, having been evaluated, is passed as a parameter to the function.

(The exceptions to this are the old versions of the TYPE, LENGTH and substring functions.)

There may be any number of arguments from 0 upwards, where each argument should be separated by a comma, except for trailing arguments with null values.

If a value for an argument is not given but values for one or more following arguments are given then the null value must be represented by a comma, for example, when the second of these arguments is a null value, the function should be written:

function-name (A, , C)

Where there are no arguments, the function must be written:

function-name ()

Each function must return a single answer in less than 256 bytes back to the executive routine.

All arguments are subject to the rules of Full Evaluation.

A blank argument must be enclosed in quotes, for example:

FUNCT (A, ' ', B)

is valid but the following:

FUNCT (A, , B)

is invalid.

Ambiguity arises if you declare a variable or function with the same name as a built-in function. To resolve this ambiguity the following order of precedence is used:

- User-defined function
- Built-in function
- Variable

It is good programming practice to avoid such ambiguities.

To force any apparent function name to be treated as a variable, place the variable in string delimiters. For example:

```
LOCAL TYPE      /*define a variable called TYPE
X = TYPE(1)      /* interpreted as built-in function
X + 'TYPE(1)'    /* interpreted as variable
```

User-defined Functions

A user-defined function is a function written by users in a language other than the procedures language. The languages available are any language capable of handling the argument table passed by the executive routine.

All function names must conform to the same naming rules as those applied to user-defined variable names, except that function names must not exceed 8 characters in length.

Before each user-defined function can be executed the name of that function must be specified by adding an entry in the source module MPLUF. For further details see your Manager Products installation manual. If a user-defined function has not been specified within MPLUF then it will not be recognized by the software as a function and instead an attempt will be made to interpret it as a user-defined variable.

COBOL Argument Block

The following specification is supplied in member EFABCOB in dataset MP.SOURCE.

Figure 1 • COBOL Argument Block

```
01 EFAB.
02 EFABANLN.
    04 EFABNAME          PIC X(8) .
    04 EFABWORK          OCCURS 18 TIMES
                          PIC S9(9)  COMP SYNC.
    04 EFABRETC          PIC X.
    88 EFABRCO           VALUE "0".
    88 EFABRC1           VALUE "1".
    88 EFABRC2           VALUE "2".
    04 EFABANSL          PIC S9(9)  COMP SYNC.
    04 EFABANST          PIC X.
```

```
      88 EFABVTAN          VALUE "C".
      88 EFABVTNM          VALUE "N".
      88 EFABVTNL          VALUE "U".
      04 EFABANSV          PIC X(255).
      04 EFABANSB          PIC S9(9)    COMP SYNC.
      04 EFABARGN          PIC S9(9)    COMP SYNC.
02 EFABAFLN              OCCURS 0 TO 122 TIMES
                        DEPENDING ON EFABARGN.
      04 EFABARGL          PIC S9(9)    COMP SYNC.
      04 EFABARGT          PIC X.
      88 EFABATAN          VALUE "C".
      88 EFABATNM          VALUE "N".
      88 EFABATAO          VALUE "0".
      88 EFABATNL          VALUE "U".
      04 EFABARGV          PIC X(255).
      04 EFABARG8          PIC S9(9)    COMP SYNC.
```

PL/I Argument Block

The following specification is supplied in member EFABPLI in dataset MP.SOURCE.

Figure 2 • PL/I Argument Block

```
DCL
1 EFAB                      BASED (EFAB_PTR) .
  3 EFABANLN.
    5 EFABNAME              CHAR (8) ALIGNED,
    5 EFABWORK (18)         FIXED BIN (31) ALIGNED,
    5 EFABRETC              CHAR (1),
    5 FILLER00001           CHAR (3),
    5 EFABANSL              FIXED BIN (31) ALIGNED,
    5 EFABANST              CHAR (1),
    5 EFABANSV              CHAR (255),
    5 EFABANSB              FIXED BIN (31) ALIGNED,
    5 EFABARGN              FIXED BIN (31) ALIGNED,
  3 EFABAFLN (EFABAFLN_REFER REFER (EFABARGN)) ALIGNED,
    5 EFABARGL              FIXED BIN (31),
    5 EFABARGT              CHAR (1),
    5 EFABARGV              CHAR (255),
    5 EFABARG8              FIXED BIN (31);
```

When the package of functions is linked, PL/I functions must not be included. All PL/I functions should be left as unresolved external references.

Assembler Argument Block

The following specification is supplied in member EFABBAL in dataset MP.SOURCE.

Figure 3 • Assembler Argument Block

```

*
*   PROCEDURES LANGUAGE EXTERNAL FUNCTION ARGUMENT BLOCK
*
EFABNAME DS          CL8          FUNCTION NAME
EFABWORK DS          18F          WORK AREA
*   RESPONSE FIELDS
EFABRETC DS          X            RETURN CODE
EFABRCO EQU          C'0'        NORMAL COMPLETION
EFABRC1 EQU          C'1'        ERROR + MESSAGE SUPPLIED
EFABRC2 EQU          C'2'        ERROR + STANDARD MESSAGE
EFABANSL OS          F            OPTIONAL ANSWER LENGTH
EFABANST DS          X            MANDATORY ANSWER TYPE
EFABVTAN EQU          C'C'        TYPE IS ALPHANUMERIC
EFABVTNM EQU          C'N'        TYPE IS NUMERIC
EFABVTNL EQU          C'U'        TYPE IS NULL
EFABANSV DS          CL255        EITHER/ ANSWER VALUE (CHARACTER)
EFABANSB DS          F            OR ANSWER VALUE BINARY
EFABARGN DS          F            NUMBER OF ARGS
EFABANLN EQU          *.EFABNAME  LENGTH OF MANDATORY FIELDS
*   ARGUMENT FIELDS
*   THE FOLLOWING BLOCK WILL BE REPEATED. 1 PER ARG
EFABARGE EQU          *
EFABARGL DS          F            ARG LENGTH
EFABARGT DS          X            ARG TYPE
*   ARG TYPE EQUATES ARE AS FOR EFABANST ABOVE. PLUS
EFABVTAO EQU          C'O'        ARG WAS ADMITTED
EFABARGV DS          CL255        ARG VALUE (CHARACTER)
EFABARGB DS          F            ARG VALUE (BINARY)
EFABAFLN EQU          *.EFABARGE  LENGTH OF 1 SET OF ARG FIELDS

```

General Notes on Argument Blocks

On return from the user-defined function, EFABRETC is examined.

If EFABRETC contains 2 (or in fact anything else except 0 or 1), the executive routine terminates, outputting a standard function failure message.

If EFABRETC contains 1, the same things happen, except that the standard function failure message is replaced by an error message expected to have been passed in EFABANSV. This message may be up to 50 characters in length. The length may be explicit (in EFABANSL) or implicit.

If EFABRETC contains 0, the user-defined function coding is expected to have returned a value in the EFABANSx fields. These fields are processed according to the following rules.

- If EFABANST contains 'U', all other answer fields are ignored:
- If EFABANST contains 'N', then
 - If EFABANSV contains nulls, the value in EFABANSB is used (a null value is taken as zero)
 - If EFABANSV does not contain nulls, then
 - If EFABANSB contains nulls, the value in EFABANSV is used
 - If EFABANSB does not contain nulls, the value in EFABANSB is used
- If EFABANST contains 'C', then
 - If EFABANSV contains nulls, processing proceeds as if EFABANST had contained 'U'
 - If EFABANSV does not contain nulls, the value in EFABANSV is used

After it has been decided to use EFABANSV according to the rules above, the length field EFABANSL is examined. If it contains nulls or a value in excess of 255, the length of the data in EFABANSV is determined by the position of the rightmost non-null byte. If EFABANSL contains a value between 1 and 255, that value is taken to be the length of the data in EFABANSV.

Outputting Information

The WRITEL and SAY directives are used to output textual information from an executive routine. This information is always output to the Primary Output Device.

The SET TRACE command and the TRACE and VLIST directives are used to output debugging information. TRACE and VLIST output is either to the Primary Output Device or to the Secondary Output Device if one is specified.

By default, SET TRACE output is directed to a ddname of MPTRACE. This default may be changed and output can be directed to the terminal in on-line use. Refer to [Chapter 10, "Debugging," on page 193](#) for further details of the SET TRACE command.

The WRITEF directive is the same as the SAY directive except that it outputs information to a specified USER-MEMBER, if one has been declared with the SENDF command. The CLOSEF command can be used to reset or change the target user member.

Manipulating Buffers

Executive routines can be set up to manipulate output from commands in two ways:

- The output from the command can already be displayed on the screen in the current buffer. If you then execute an executive routine, it can automatically read the contents of this current buffer using the `&CURL` system variable.
- You may wish the executive routine itself to issue the command. In this case if `OUTPUT-EDIT` is set to `ON` then the output from the command can be referenced as before using the `&CURL` variable.

You may choose to suppress output to the screen until the executive routine has terminated. This is achieved by the `SET EXEC-WRITE OFF` command. This allows the executive routine to replace the original output by a reformatted version of the output without the original version being displayed on the screen.

Your executive routine may also be used to generate `UPDATE` or `EDIT` buffers within executive routines. As you are explicitly generating a new buffer it is neither necessary nor possible to generate an `UPDATE` buffer or an `EDIT` buffer with `SET OUTPUT-EDIT ON`.

Debugging

There are several directives and commands that are useful in developing and debugging executive routines.

You can use the:

- `VLIST` directive to display the contents of variables
- `TRACE` directive to display selected information as an executive routine executes
- `SET TRACE` command to display trace information without embedding `TRACE` directives in selected procedures
- `SET ECHO ON` command to display each command before it is executed
- `SIGNAL ON ERROR` directive to trap error conditions generated by commands
- `SIGNAL ON SYNTAX` command to trap error conditions generated by directives

Output from the `TRACE` and `VLIST` directives goes to the Secondary Output Device (if there is one). By default, output from the `SET TRACE` command goes to a ddname of `MPTRACE`; you can select an alternative ddname.

Parsing

Parsing is the process by which a string is divided into its component parts, its words. Parsing is used in:

- The assignment of parameter variables
- The PARSE directive
- The WORD family of functions

The string delimiters and undelimited spaces in a string define the words in that string. A delimited space is counted as part of a word, it does not define words. String delimiters must be paired; the first in the pair is the left delimiter and the second is the right delimiter.

String delimiter characters are set in your Manager Products environment. They consist of the following:

- ' (single quote)
- " (double quote)
- Any other characters set as string delimiter characters by your administrator

To see your current string delimiter characters, enter:

```
QUERY STRING-DELIMITER ;
```

There are four parsing methods: 1 to 4. Method 4 is the default, and will be all many users ever need. To switch parsing methods use the PARSEOPTION directive. To test if a string can be parsed under the current parsing method use the PARSABLE function.

String delimiter characters are not processed as string delimiters in all parsing methods. The parsing methods are described below.

Method 1

String delimiter characters are processed as string delimiters. String delimiters delimit spaces and define words.

Method 2

String delimiter characters are processed as string delimiters. String delimiters delimit spaces but do not define words.

Method 3

String delimiter characters are processed as ordinary characters.

Method 4

String delimiter characters are only processed as string delimiters selectively. A string delimiter character

- As first character in the string, or
- Preceded by a space

is processed as a left delimiter, and there must be a corresponding right delimiter. String delimiters delimit spaces and define words. Other occurrences of string delimiter characters are processed as ordinary characters.

Example

Consider the string:

```
11111 ' '22 333
```

Under method 1 the string consists of four words:

```
11111  
' '  
22  
333
```

Under method 2 the string consists of two words:

```
11111 ' '22  
333
```

Under method 3 the string consists of three words:

```
11111 '  
'22  
333
```

Under method 4 the string cannot be parsed since it contains a left delimiter, but no right delimiter.

Efficiency And Readability

If you make your executive routines efficient, more of the machine's resources will be free for you and other users. If you make your executive routines readable, they will be easier to maintain. Here are a few tips for writing efficient and readable executive routines:

- Use meaningful variable names, not ampersand variables
- Erase array variables or groups of variables if you finish with them part way through the executive routine, using the DROP or RELEASE directives or DRELEASE command
- Define a literal delimiter using MPXX LITERAL=
- Prefix commands by the MPR or MPRE directive
- Enclose literals in literal delimiters
- Use /* to define comments
- Use the DO, ITERATE, LEAVE, IF, CALL, and RETURN directives, not the GOTO directive
- Indent instructions so that it is easy to see where instruction blocks begin and end
- Use the concatenation operator ||, and not implicit concatenation
- Consider retaining frequently called executive routines in virtual storage, using the SET EXECUTIVE-RETENTION command and RETAIN directive
- Preserve your environment, if necessary, using the PUSH and PULL commands

3

Setting Up And Using Executive Routines

This chapter contains these sections:

Introduction	21
Corporate Executive Routines	22
Setting Up EXECUTIVE-ROUTINE Members	23
User And Transient Executive Routines	24
Executing Executive Routines From The Command Area	24
Line Commands	24
Line Commands Example	25
Cursor Spatial Commands	26
Cursor Spatial Commands Example	27
Using Prefix Commands/Resolving Name Conflicts	27

Introduction

There are three types of executive routine:

- Corporate
- User
- Transient

Corporate executive routines are maintained by the systems administrator, but can be executed by general users.

Users can set up their own executive routines by storing them in USER-MEMBERS and TRANSIENTs and subsequently executing them as user executive routines and transient executive routines respectively.

To list executive routines use the following commands:

- MP-AID LIST EXECUTIVES;
- MP-AID LIST USER-MEMBER;
- MP-AID LIST TRANSIENT;

To print executive routines use the following commands:

- MP-AID PRINT EXECUTIVE member-name;
- MP-AID PRINT USER-MEMBER member-name;
- MP-AID PRINT TRANSIENT member-name;

Executive routines can be used in three different modes of operation:

- From the command line
- As line commands
- As cursor spatial commands.

There will be name conflicts if, for example, you wish to execute an executive routine having the same name as a Manager Products command. Such name conflicts can be resolved using the prefix commands described in ["Using Prefix Commands/Resolving Name Conflicts" on page 27](#).

Corporate Executive Routines

Executive routines are stored as EXECUTIVE members on the MP-AID by the systems administrator. The contents of these members may subsequently be executed as corporate executive routines by general users, subject to any access controls.

EXECUTIVE members are initially defined as EXECUTIVE-ROUTINE members on the Administration Dictionary by the systems administrator. EXECUTIVE-MEMBERS can then be constructed on to the MP-AID as EXECUTIVE members.

When a user executes a corporate executive routine only the following commands can be executed:

- All commands normally available to the user
- Any non-restricted commands that have been disabled
- Any restricted SET commands.

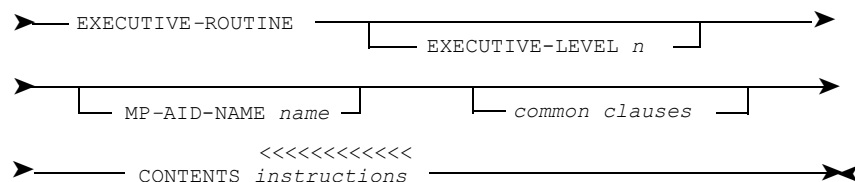
No other restricted commands can be executed.

With the Systems Administrator's Environmental Control Facility, the following additional features are provided:

- Access to a corporate executive routine may be restricted by the systems administrator to users having the appropriate access level
- Access levels are defined in a Logon Profile and are matched against the access level defined for a corporate executive routine.

Setting Up EXECUTIVE-ROUTINE Members

The syntax of EXECUTIVE-ROUTINE repository members is as follows:



The EXECUTIVE-LEVEL clause is optional, and can only be used if the systems administrator's Environmental Control Facility is installed. If the clause is used, it must contain a value from 0 through 255.

The EXECUTIVE-LEVEL permits the systems administrator to restrict access to those users whose Logon Profile allows access to this executive level.

The MP-AID-NAME clause is optional. If present, it must not exceed 10 alphanumeric characters and may be optionally delimited.

Repository member names can be up to 32 characters long. However, MP-AID member names cannot exceed 10 alphanumeric characters. The optional MP-AID-NAME clause enables you to specify a member name under which the corporate executive routine can be constructed onto the MP-AID, when the repository member name is unsuitable. An EXECUTIVE-ROUTINE member with a name longer than 10 characters cannot be constructed onto the MP-AID, unless it contains a valid MP-AID-NAME clause.

User And Transient Executive Routines

With the Extended Interactive Facility you can create USER-MEMBER and TRANSIENT members. You can execute USER-MEMBER members as user executive routines, and TRANSIENT members as transient executive routines.

If you wish to execute a USER-MEMBER as a user executive routine it is essential for the MPXX directive to be inserted in column 1 on the first line. This is not necessary for executing a TRANSIENT member as a transient executive routine.

User executive routines and transient executive routines can contain any valid Procedures Language statements.

Unlike Corporate executive routines, neither Manager Products commands that have been disallowed by the systems administrator nor restricted SET commands can be included in user executive routines or transient executive routines.

User executive routines and transient executive routines can only be executed by the user who created them, or another user having the same Logon Identifier.

When a user or transient executive routine is executed and then modified during the same executive routine run, the modifications will not be acted upon until after the current highest level executive routine has terminated.

You can only execute transient executive routines using the prefix command TRANSIENT-EXECUTIVE (see ["Using Prefix Commands/Resolving Name Conflicts" on page 27](#)).

Executing Executive Routines From The Command Area

Enter the name of the executive routine in the Command Area followed by the values of any parameters. For example, if you wish to execute an executive routine called SUBSTITUTE with parameter values DISPLAY and 12, then enter:

```
SUBSTITUTE DISPLAY 12
```

Line Commands

Enter the name of the executive routine in the Line Command Area as a Line Command. The values of any parameters are taken from the associated data line. That is, the value of parameter &PO is taken from the leftmost element of the line, the value of parameter &P1 is taken from the next element to the right, and so on.

Unless within a delimited string, the end of an element is assumed when a space character is encountered. As a result, multiple elements which represent a certain parameter may need to be displayed as a delimited string. If delimiters are not present, when required, then incorrect results may occur.

Once a Line Command has been entered in the Line Command Area, it is only necessary to enter an asterisk (*) on any following lines in order to repeat the command.

The name of an executive routine used as a Line Command must not exceed 5 characters and must not include any '?' characters.

Line Commands Example

Assuming you want to PRINT and REPORT particular members whose names appear on the screen in output resulting from a previously issued LIST command. You can set up an executive routine (called, for example, LC1) so that it can be entered as a Line Command.

LC1 should contain the following instructions:

```
MPXX
/* THIS IS AN EXAMPLE OF A LINE COMMAND ;
PRINT &PO ;
REPORT &PO ;
```

To use LC1 as a Line Command, enter LC1 in the Line Command Area on the same line as the name of the member to be printed and reported. Press Enter to invoke the command, the value of the parameter &PO is set to the contents of the first element in the associated data line, which in this case is a member name.

Note:

After entering LC1 in the Line Command Area on one line, enter an asterisk (*) in the Line Command Area of any following lines if a print and report is required of more than one member.

[Figure 4](#) shows LC1 being used on three members which are included in output from a previous LIST command:

Figure 4 • LC1 Executive Routine

LIST OF MEMBERS							
MEMBER NAME	TYPE	USAGE	CONDITION	AC	ALT	REM	
HEAD-LINE-1	GROUP	0	SCE ENC	0	0	0	LC1==
HEAD-LINE-2	GROUP	0	SCE ENC	0	0	0	=====
EMP- IDENT	GROUP	0	SCE ENC	0	0	0	*=====
REC-EMP-ABS	GROUP	0	SCE-ENC	0	0	0	=====
SYS-EMP-REC	SYSTEM	0	SCE-ENC	0	0	0	*=====
SYS-EMP-HIST	SYSTEM	0	SCE-ENC	0	0	0	=====

```
LIST CONTAINS      4 GROUPS
                   2 SYSTEMS
                   6 MEMBERS IN TOTAL

===>
```

In [Figure 4 on page 25](#), the members HEAD-LINE-1, EMP-IDENT and SYS-EMP-REC are printed and reported.

Cursor Spatial Commands

A Cursor Spatial Command is an executive routine that takes a parameter from an element of text or a delimited string at the cursor position on the screen. The element or string can be referenced within the executive routine by using the system variable &CURS.

There are two ways in which you can make an element or string available to an executive routine in &CURS:

Method 1

- Assign a PF key to the name of an executive routine so that it will be invoked whenever that PF key is pressed. For example entering SET PF11 PROCL-ONE ; causes the executive routine PROCL-ONE to be executed whenever PF11 is pressed
- Place the cursor at the element or string on the screen to be supplied as a parameter to the executive routine.
- Press the PF key which is assigned to the name of the executive routine to execute that executive routine.

Method 2

- Insert the name of the executive routine onto the Command Line without pressing Enter.
- Move the cursor to the element or string on the screen which is to be supplied as a parameter to the executive routine.
- Press Enter.

The start and/or end of an element of text is defined by a comma or space. &CURS returns the individual element under which the cursor is placed, if the element is not part of a larger delimited element. Leading or trailing commas are not returned. If the cursor is placed under a comma, then the comma and immediately following characters are returned.

If the cursor is placed within a delimited string, then &CURS returns the whole string except for the delimiters, for example:

```
'EMP CODE'          gives      EMP CODE
```

If the cursor is placed under either of the delimiters, then &CURS returns the whole string including the delimiters, for example:

```
'EMP CODE'          gives      'EMP CODE'
```

Cursor Spatial Commands Example

Set up the following commands in an executive routine named CS1:

```
MPXX
/* This is an example of a Cursor Spatial Command
PRINT &CURS ;
REPORT &CURS ;
```

Set up PF key 15 using the following command:

```
SET PF15 IMMEDIATE CS1 ;
```

CS1 can then be used to obtain a PRINT and REPORT of any member-name which is displayed. Place the cursor under any character of the required member name and press the PF15 key.

Using Prefix Commands/Resolving Name Conflicts

This is the search sequence for the commands:

- Primary Commands
- User executive routines
- Corporate executive routines

So, for example, if you have an executive routine with the same name as a Primary Command, you will not be able to execute it unless you use one of the prefix commands described in the following table.

To execute a transient executive you must always use the TRANSIENT-EXECUTIVE prefix command.

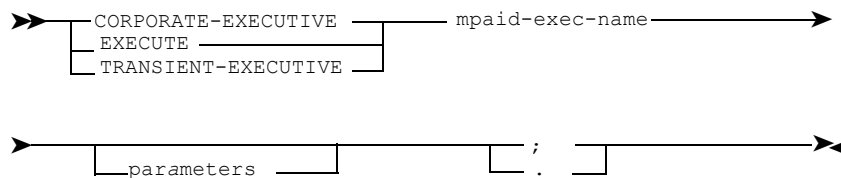
In order to modify the search sequence or select a specific executive type, use these prefix commands:

Prefix Command	Search Sequence
CORPORATE-EXECUTIVE	Corporate Executive only
EXECUTE	User executive then Corporate Executive
TRANSIENT-EXECUTIVE	Transient executive only

They have these short forms:

CEXEC
EXEC
TEEXEC

The syntax is as follows:



Note: _____

It is not possible to use these prefix commands if:

- An executive routine is entered as a Line Command
- The name of an executive routine is preceded by the TRANSFER directive.

In such cases all you can do to resolve the name conflict is to rename one of the executive routines.

4

Variables

This chapter includes these sections:

Variables and Parameters	30
User-defined Variables	31
Arrays	31
ASG-defined Variables	33
System-assigned Variables	33
User-assigned Variables	33
Command Variables	33
Profile Variables	34
Global Variables	34
Local Variables	35
Installation Variables	36
Parameter Variables	36
System Variables	37
&BUFN	37
&CCOD	37
&CCOL	37
&COLO	38
&CROW	38
&CURL	38
&CURS	38
&DATE	38
&DICT	38
&ECOD	39
&ENAM	39
&ENVO	39
&ENVM	39
&ENVT	40
&LINC	40
&LINO	40
&LOGO	40
&MODE	41

&MSLN.....	41
&MSLV.....	41
&MSNO	41
&MSTX.....	41
&PNUM	41
&PVAL.....	42
&SCOD.....	42
&STAT	42
&TIME	42
&TRMC	42
&TRMR	42
&USER.....	42
Return Codes.....	42

Variables and Parameters

Variables are assigned a value using an instruction of this format:

variable-name = expression

For example:

ORDER_DATE = ISSUE_DATE+42

There are two basic types of variables available with the procedures language:

- User-defined variables (that is, variables with user-defined names such as COUNTER)
- ASG-defined variables (that is, variables with ASG-defined names such as &L7).

In many circumstances you have the choice of using either type of variable. ASG recommends that you use user-defined variables where possible, as it will help to improve the intelligibility of your executive routines.

All variables may contain text strings of 0 to 255 characters. Variables set to an integer in the range -2147483648 to 2147483647 inclusive are eligible for arithmetical operations.

User-defined Variables

User-defined variable names can be up to 50 characters long. This allows you to give variables meaningful names.

Command and Profile variables are only available as user-defined variables. Command variables retain their value until the termination of the currently-running highest-level executive routine. Profile variables retain their value at LOGOFF under specified conditions (see ["Profile Variables" on page 34](#)).

If you use a variable without declaring it, it is declared as local. Local variables can be explicitly declared using the LOCAL directive.

The GLOBAL, COMMAND and PROFILE directives declare a variable to be a global, command or profile variable respectively.

User-defined variables may be removed using the DROP or RELEASE directives.

A variable may be declared as local even if it is already declared as a command or global variable, in which case the local variable takes precedence until it is removed.

The name of a user-defined variable must conform to these rules:

- Names must be 1 to 50 characters in length
- Names can consist of the following characters: letters, digits, @ (at), # (hash), £ (pound) or _ (underscore)
- Names must begin with a letter.

ASG recommends that you do not use names that are identical to directive keywords such as EQ, OR and THEN, as this may cause the executive routine to fail to execute as desired.

Arrays

An array is a variable with more than one element. An array element is referenced by following the array name by an expression in brackets. Elements are numbered from 1 to 999,999,999.

Array subscripts are evaluated according to the rules of Full Evaluation.

You can assign values to array elements using an instruction of the following format:

```
variable-name(expr) = value
```

where *expr* is or evaluates to an integer in the range 0 to 999,999,999.

The list following shows the ways of referring to array elements:

USERVAR	The first element in the array USERVAR
USERVAR(1)	The first element in the array USERVAR
USERVAR(6)	The 6th element in the array USERVAR
USERVAR(expr)	The <i>n</i> th element in the array USERVAR, expression <i>expr</i> evaluates to the integer value <i>n</i>
USERVAR(3)(2,1)	The substring (start position 2, length 1) of the third element of USERVAR
USERVAR(2,1)(3)	Interpreted as USERVAR(1)(2,1) (3).

There must be no spaces between the array name and the opening bracket. When a particular array element is not specified the first is assumed, for example:

```
USERVAR ( 6 ) = X
```

sets the 6th element of USERVAR to the value of the first element of X.

Element zero of an array has a special purpose: its value is returned whenever a null element of the array is referred to in an expression.

Element zero of an array can only be assigned to. In all other circumstances it is not considered an array element. For example, its value cannot be directly retrieved.

You can set one array equal to another. For example:

```
TARGET ( ) = SOURCE
```

sets every element of the array TARGET to the corresponding element of the array SOURCE.

In instructions of this form the right hand side must be the name of an array, not an expression that evaluates to the name of an array.

ASG-defined Variables

There are two types of ASG-defined variables:

- System-assigned
- User-assigned.

System-assigned Variables

System-assigned variables are maintained by Manager Products. Your executive routines can only read them, not assign values to them. They provide information on the Manager Products and system environment.

User-assigned Variables

User-assigned variables can be assigned by the user. They can be subdivided into the following types:

- Global variables
- Local variables
- Installation variables
- Parameters.

The ampersand character (&) indicates an ASG-defined variable and consequently the procedures language expects to read G, I, L, P or one of the system variable names following the ampersand. If you wish to use & as a literal you must use either:

- &&
- Literal delimiters. For example, if the hash mark (#) has been defined as a literal character, you would write #&#.

Command Variables

There are no predefined command variables. You define all command variable names.

Command variables exist from their declaration until the end of the highest-level executive routine.

You use command variables when you wish to share information between executive routines, but you do not want the information to continue to exist after the executive routines have finished.

Profile Variables

There are no predefined profile variables. You define all profile variable names.

User defined profile variables are identical to global variables except that they are not lost at logoff but retained in a Variable Pool member on the MP-AID and automatically restored at the next logon provided that:

- The MP-AID is updateable
- The user is logged on under an exclusive logon or is the systems administrator.

Any user may define profile variables, but they will only be saved between Manager Products sessions under the circumstances noted above.

Global Variables

&G0 to &G99 are the predefined global variables.

Global variables are available to any executive routine called during a Manager Products session. Their values are maintained from one executive routine to the next and may be reassigned by any executive routine. Global variables are set to null when the session starts.

Here is an example of the use of global variables. This example sets up two executive routines, START and FINISH.

The START executive routine contains the following commands:

```
MPXX
&G0 = &DATE
&G1 = &TIME
```

The FINISH executive routine contains the following commands:

```
MPXX LITERAL=#
MPR SWITCH OUTPUT TO MPRT;
MPR SET ECHO ON;
MPR KEEP AND LIST IF AMENDED AFTER "&G0" AT "&G1" BY &USER;
MPR PERFORM 'PRINT "*" ' KEPT;
MPR SET ECHO OFF;
MPR SWITCH OUTPUT TO MPOUT;
SAY #SESSION RECORD WAS OUTPUT VIA MPRT#
```


If the START executive routine is invoked when a user logs on (for example, as part of the Logon Profile), the value of global variable &G0 will be set to the current date, and the value of global variable &G1 will be set to the time that START executed. These values will be maintained for the duration of the session, provided that these global variables are not reassigned elsewhere.

If the user, immediately before logging off from Manager Products, invokes the executive routine FINISH, a LIST and PRINT of all members amended during the current session will be written out to an external file, as a session record. The values of global variables &G0 and &G1, which have been maintained whilst the user is logged on, will provide the selection criteria for the FINISH executive routine.

Local Variables

&L0 to &L99 are the predefined local variables.

Local variables are available for the exclusive use of an individual executive routine. They are initially set to null on entry to the executive routine.

Here is an example of the use of local variables. This example sets up executive routine ARC-PRINT. ARC-PRINT contains the following commands:

```
MPXX
&L2 = &STAT
MPR STATUS ARCHIVE;
MPR PRINT &P0;
MPR STATUS &L2;
```

In the above executive routine, the current status is assigned to local variable &L2. After the PRINT command has been performed, local variable &L2 is then used to switch the user back to the current status.

ARC-PRINT may be used to print a member from the ARCHIVE status, without having to manually enter STATUS or PRINT commands.

The following command could be used to obtain a print of a member called 1992-TAX-CODES, which is stored in the ARCHIVE status:

```
ARC-PRINT 1992-TAX-CODES;
```

Installation Variables

&I0 to &I99 are the only installation variables.

These can only be assigned by the systems administrator in an executive routine called from GLOBAL0000, and may not be altered by any user. The assigned values may be read within any executive routine.

A use for installation variables might be to provide access to standard terms used within an organization when setting up transient executive routines and user executive routines.

Parameter Variables

Each executive routine has 100 parameter variables: &P0 to &P99.

When an executive routine is invoked, the input line is parsed and the words are assigned to parameter variables, starting from &P0. Within the executive routine parameter variables can be used exactly as if they are local variables.

Here is an example of the use of parameter variables. If you enter the command:

```
MYEXEC DAVE 10;
```

the executive routine MYEXEC is invoked with &P0 initialized to DAVE and &P1 to 10. All other parameter variables are initialized to null.

Parameter values are delimited by spaces. So if a parameter value has a space in it, you must enclose the value in string delimiters. For example:

```
MYEXEC 'DAVID S' 10;
```

These string delimiters are stripped out when the parameter variables are set.

An executive routine call in an executive routine is subject to Limited Evaluation.

So in the following example:

```
A = 'A'
B = ''
C = 'C'
EXEC1 A B C ;
```

the expressions A B C evaluate to 'A C'. So the command:

```
EXEC1 A C;
```

is executed. There are two parameters. To prevent this, enclose potentially null parameters in string delimiters. For example:

```
EXEC1 A 'B' C;
```

There are now three parameters.

System Variables

The system variables available with the procedures language are described below in alphabetical order. System variables are ASG-defined and system-assigned.

&BUFN

Buffer name, if any, of the currently addressed buffer. There are five different types of buffer. You can find out the current buffer type from the system variable &MODE.

Buffer Type	&BUFN
Command Mode	Null
InfoBank Mode	Name of the InfoBank panel being viewed
Lookaside Mode	Null
Edit Mode	Name of the USER-MEMBER being edited (if any), otherwise null
Update Mode	Name of the repository member being updated (if any), otherwise null

The currently addressed buffer is determined by use of the SET OUTPUT-EDIT ON/OFF command.

&CCOD

Return code from the previous command or executive routine.

&CCOL

Offset from the start of a data line to the character under which the cursor is placed. The value of &CCOL is set to 0 for the first character, 1 for the second character and so on. If the cursor is outside the current buffer; that is, in the:

- Line Command Area,
- Command Area, or
- Heading Area

then &CCOL is set to -1.

&CCOL is generally used in conjunction with the system variable &CROW to determine the cursor position.

&COLO

Offset of the first visible column of text at the left margin in the current buffer. If the buffer has not been offset due to a previous RIGHT command then the offset given is 0. Otherwise the value given is the number of columns of text before the left margin which are not displayed due to this offset. The currently addressed buffer is determined by use of the SET OUTPUT-EDIT ON/OFF command.

&CROW

Number of data lines from the *** TOP OF DATA *** line to the data line where the cursor is placed. The value of &CROW is set to 1 for the first data line, 2 for the second data line and so on. If the cursor is outside the current buffer; that is, in the:

- Line Command Area,
- Command Area, or
- Heading Area

then &CROW is set to -1.

&CROW is generally used in conjunction with the system variable &CCOL to determine the cursor position.

&CURL

Contents of current line in the currently addressed buffer. The currently addressed buffer is determined by use of the SET OUTPUT-EDIT ON/OFF command.

&CURS

Value of an element of text or a delimited character string at the cursor position.

&DATE

Current date (in the format defined in the DCUST installation macro).

&DICT

Name of repository currently open (if any).

&ECOD

Highest return code from executive routines or commands executed in the current executive routine.

&ENAM

Name of current executive routine.

&ENVO

Operating System in use. &ENVO is one of these values:

- D DOS
- M ManagerView
- O OS
- P Programmable Work Station (PWS)
- S BS2000
- V VM/CMS
- W Web client

&ENVM

Environment in use. &ENVM is one of these values:

- A Access call
- F Full-screen interactive
- L Line-mode interactive
- M ManagerView
- P Programmable Work Station (PWS)
- S Standard environment (batch)
- W Web client

&ENVT

TP Monitor in use. &ENVT is one of these values:

- A Access Call Interactive (when any other setting is not appropriate)
- B Batch
- C CICS
- D IMS/DC
- F ICCF
- I TSO/ISPF
- M ManagerView
- K Background TSO invoked via IKJEFT01
- P Programmable Work Station (PWS)
- R ROSCOE
- S Siemens Timesharing Interface (TIAM)
- T TSO
- V VM/CMS
- W Web client

In MVS TSO/ISPF environments, the value I is returned only when Manager Products requires ISPF to be present (that is, when using selectable unit FE70). When ISPF is not required (selectable unit TP7), the value T is returned.

&LINC

Total number of lines in the currently addressed buffer. The currently addressed buffer is determined by use of the SET OUTPUT-EDIT ON/OFF command.

&LINO

Number of data lines from the *** TOP OF DATA *** line to the current line in the currently addressed buffer. The value of &LINO is set to 1 for the first data line, 2 for the second data line and so on. The currently addressed buffer is determined by use of the SET OUTPUT-EDIT ON/OFF command.

&LOGO

Logon Identifier.

&MODE

Buffer mode of currently addressed buffer. &MODE is set to one of these values:

- C Command Mode
- E Edit Mode
- I InfoBank Mode
- L Lookaside Mode
- U Update Mode.

The currently addressed buffer is determined by use of the SET OUTPUT-EDIT ON/OFF command.

&MSLN

Complete message line of the last Manager Products message.

&MSLV

Severity level of the last Manager Products message. &MSLV is set to one of the following values:

- C Critical error message.
- E Error message
- I Informatory message
- S Serious error message
- W Warning message

&MSNO

Number of the last Manager Products message (leading zeroes are not present).

&MSTX

Text of the last Manager Products message.

&PNUM

Number of input parameters supplied to the current executive routine, that is, the number of &P variables that were assigned values when the executive routine was called.

&PVAL

Content of all parameters supplied to the current executive routine, exactly as entered by the user.

&SCOD

Highest return code so far of the current Manager Products session.

&STAT

Name of the current status (if any).

&TIME

Current time (in the format defined in the DCUST installation macro).

&TRMC

Number of columns on the terminal screen. This is a physical characteristic of the type of terminal in use.

&TRMR

Number of rows on the terminal screen. This is a physical characteristic of the type of terminal in use.

&USER

Name of the current repository user (if any).

Return Codes

Return codes are set after each Manager Products command or executive routine depending on the maximum severity level of all messages issued during execution of that command or executive routine as follows:

- Result code 0 is issued for Informatory (I level) messages
- Result code 4 is issued for Warning (W level) messages
- Result code 8 is issued for Error (E level), Serious (S level), or Critical (C level) messages

The following system variables provide information regarding return codes:

- &CCOD** Return code from the previous executive routine or command
- &ECOD** Highest return code from executive routines or commands executed in the current executive routine
- &SCOD** Highest return code so far of the current Manager Products session.

If an executive routine does not return using the EXIT directive then:

EXIT &ECOD

is assumed.

&CCOD, &ECOD and &SCOD can be reset using the EXIT directive.

5

Expressions

This chapter includes these sections:

Introduction	45
Full Evaluation	47
Limited Evaluation	48
Character Expressions	49
Numeric Expressions	49

Introduction

Expressions represent those parts of instructions that may be evaluated. Any expression consists of one or more terms optionally interspersed with one or more operators.

A term is the smallest element of the procedures language that represents a distinct and separate value. It can be used alone or in combination with other terms to form an expression. These are examples of terms:

```
&L17  
ABC  
100  
POS('t','state')
```

An operator is one of these symbols:

	Concatenation
+	Addition
-	Subtraction
*	Multiplication

/ Integer division giving quotient
! Integer division giving remainder
** Raise to a power

The concatenation operator concatenates two terms.

For example:

```
literal #  
&L1 = #NAME#  
&L3 = &L1 | |8
```

results in &L3 being set to the string NAME8.

Note: _____

The concatenation operator is only mandatory in situations where ambiguity may arise if it is omitted. For example:

```
SAY &L2 | |23
```

However, ASG recommends that it be used between all terms that are to be concatenated.

Expressions are terminated by an undelimited space character.

There are two types of expression:

- Character
- Numeric

Expressions are evaluated strictly left to right, regardless of the type of operator. All expressions are assumed to be numeric expressions, until a non-numeric term is encountered. For example, the expression `2+3 | | FRED` results in the character expression `5FRED`. No expression may exceed 255 characters in length after evaluation, except for expressions following the `WRITEF` directive, where the limit is 32760 characters.

There are two types of evaluation:

- Limited Evaluation
- Full Evaluation

Which type of evaluation takes place depends on the context.

Full Evaluation

Full Evaluation takes place with all directives that are evaluated (apart from three that are subject to Limited Evaluation), and in:

- Assignment instructions
- Array subscripts
- Function calls.

These actions are performed with Full Evaluation:

- All characters are processed left-to-right, for example:

```
SAY 5+9ACCOUNT      gives      14ACCOUNT
```

- Concatenation symbols are removed

```
SAY ABCDE||12345    gives ABCDE12345
```

- Literal delimiters (as defined by LITERAL) are removed, character strings within literal delimiters are then returned unchanged. For example:

```
LITERAL #
&GO = 45
SAY #&GO BOOK#      gives      &GO BOOK
```

- String delimiters are removed, character strings within string delimiters then being processed under the rules of Limited Evaluation, for example:

```
&GO = 45
SAY '&GO BOOK'      gives      45 BOOK
```

- Array subscripts and function arguments are evaluated
- All variables are replaced with their values
- Function calls are evaluated
- Arithmetic expressions are evaluated.

Limited Evaluation

Limited Evaluation takes place with the following directives:

- WRITEL
- MPR
- TRANSFER

and with expressions in any instruction that is not recognized by the procedures language, the instruction then being processed as a command, for example:

```
X Y;
```

where X and Y are user-defined variables.

Expressions subject to Limited Evaluation may not exceed 4000 characters in length after substitution, except for expressions following the WRITEL directive, where the limit is 598 characters.

These actions are performed with Limited Evaluation:

- Concatenation is performed, for example:

```
WRITEL NUMBER||42      gives      NUMBER42
```

- Any variable is replaced with its value, for example:

```
i = 1
WRITEL i                gives      1
a(1) = 1
WRITEL a(1+1-1)         gives      1
WRITEL i+a(1)           gives      1+1
```

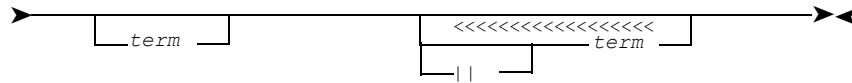
- Literal delimiters (as defined by LITERAL) are removed. The character strings within literal delimiters are not evaluated. For example:

```
LITERAL #
SAY #&GO IS CORRECT #   gives      &GO IS CORRECT
```

- For implicit substring arguments, nested implicit substrings are evaluated.

Character Expressions

A character expression is defined as:



where:

$||$ is the concatenation operator

term is any of the following:

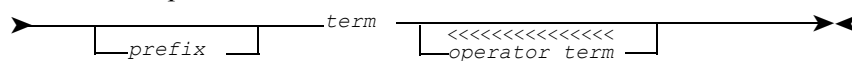
- Variable name
- Function call
- Undelimited character string with no embedded blanks
- Delimited character string

Examples

```
ABC || &PO
&P1&P2&P3
LEFT(field,4,*)
'4+2 || FRED'
```

Numeric Expressions

A numeric expression is defined as:



where:

prefix is + (plus) or - (minus)

operator is any of the following:

- | | |
|---|----------------|
| + | Addition |
| - | Subtraction |
| * | Multiplication |

/ Integer division giving quotient
! Integer division giving remainder
** Raise to a power

term is any of these:

- Variable name
- Integer
- Function

All operators have equal precedence and are processed from left to right. For example:

4+6**2 gives 100

Expressions must be in the range -2147483648 to 2147483647 inclusive, except for the second operand of ** which must be in the range 0 to 15 inclusive.

Note: _____

Null strings are not considered numeric terms.

Examples

&L2+LENGTH(range)

total-volume

1500*878

6

Example Executive Routines

This chapter contains these example executive routines:

MP-AID Copy (PROCL-01)	51
Condensed MP-AID List (PROCL-02)	52
FASTQUIT (PROCL-03)	54
Quick Sign On (PROCL-04)	54
Decimal Conversion (PROCL-05)	55
All Occurrences (PROCL-06)	56
Overlay (PROCL-07)	57
PF Key Settings (PROCL-08)	58
Compound Interest (PROCL-09)	59
ISPF Read, Write and Edit	60
ISPF Variables.	61

Some of the titles of the executive routines are followed by a repository member name in brackets. This gives the name of the member in the demo repository that contains the text of the executive routine.

MP-AID Copy (PROCL-01)

```
MPXX
/* This executive routine copies a user member to another user member.
/* The routine has three parameters: member-name1 TO member-name2.

IF &P1 ^= 'TO' THEN GOTO OUT          /* check the syntax.
MPR NOPRINT EDIT;
IF &CCOD ^= 0 THEN GOTO OUT1           /* Reject if not possible
                                          /* to open an EDIT buffer.

MPR NOPRINT GETU &P0;
IF&CCOD ^= 0 THEN GOTO OUT2           /* Did it work?
MPR NOPRINT FILE &P2;
/* Reject if destination member cannot be setup either
/* because there is not enough space available on the MP-AID
/* for another USER-MEMBER, or because a USER-MEMBER having
/* this name already exists on the MP-AID.
```

```
IF &CCOD ^= 0 THEN GOTO OUT3
WRITEL 'USER-MEMBER '&P0' HAS BEEN SUCCESSFULLY COPIED TO '&P2
EXIT

-OUT
WRITEL 'You have not entered the correct syntax.'
WRITEL 'You should have entered:'
WRITEL 'EXEC_NAME member-name1 TO member-name2'
EXIT

-OUT1
WRITEL 'Your limit of output buffers has been reached and'
WRITEL 'it is not possible to execute the command.'
EXIT

-OUT2
MPR NOPRINT XQUIT;
WRITEL 'The USER-MEMBER that you wish to copy does'
WRITEL 'not exist for your logon identifier.'
EXIT

-OUT3
MPR NOPRINT XQUIT;
IF &MSNO = 8814 THEN DO
    WRITEL 'COPY WAS UNSUCCESSFUL'
    WRITEL 'You already have a USER-MEMBER called '&P2
EXIT

-OUT3
MPR NOPRINT XQUIT;
IF &MSNO = 8814 THEN DO
    WRITEL 'COPY WAS UNSUCCESSFUL'
    WRITEL 'You already have a USER-MEMBER called '&P2
    EXIT
END
WRITEL 'It was not possible to store a copy of your'
WRITEL 'USER-MEMBER on the MP-AID due to lack of space.'
EXIT
```

Condensed MP-AID List (PROCL-02)

```
MPXX
/* This executive routine produces a condensed version of the output from
/* an MP-AID LIST member-type command. It expects one parameter: the name
/* of an MP-AID member-type.

IF arg() ^= THEN DO
    SAY 'You must give one parameter: MP-AID member-type'
    EXIT
END

MPR NOPRINT PUSH;
MPR NOPRINT SET OUTPUT-EDIT ON;
MPR NOPRINT SET EXEC-WRITE OFF;
MPR MP-AID LIST &P0;
even = ' '
odd = ' '
line_count = &LINC
/* lines_read is the line number of the output generated by the MP-AID LIST
/* command.
lines_read = 4                                /* Allow for heading and trailing blanks.
```

```

member_number = 0
SAY '+'COPIES('-',67)+'
MPR NEXT;                                /* Get the first line of output.
&L4 = SUBSTR(&CURL,1,60)
blanks = ' '
/* Re-output the first line centralized followed by header
SAY |blanks||&L4|
SAY '+'COPIES('-',67)+'
header = 'NAME          TYPE          DATE          '
SAY | header | header |
SAY '+'COPIES('-',67)+'
MPR NEXT;

-LOOP                                  /* Start of loop
MPR NEXT;
/* Check whether all output lines of output from the MP-AID LIST command
/* have been read.
IF lines_read GT line_count THEN GOTO CLEAR

/* Two lines of information, the first 32 characters from each line, are
/* read from the list of members at a time. Even numbered lines are stored
/* in even. Odd numbered lines are stored in odd. Once an even numbered line
/* has been read it is concatenated onto the end of an odd numbered line and
/* re-output. odd and even are then reset to null ready for storing the
/* next pair of lines.
/*
IF even ^= ' ' THEN DO                /* Output concatenated pair of lines
    SAY | odd | even |
    odd = ' '
    even = ' '
END
IF odd ^= ' ' THEN GOT EVEN
odd = SUBSTR(&CURL,1,32)              /* Read new odd numbered line
member_number = member_number+1
GOTO TEST

-EVEN
even = SUBSTR(&CURL,1,32)              /* Read new even numbered line
member_number = member_number+1

-TEST
lines_read = lines_read+1
GOTO LOOP                            /* Put out final lines and do the totals.

-CLEAR
blanks = ' '
IF odd EQ ' ' THEN GOTO TOTAL
IF even EQ ' ' THEN even = LEFT(blanks,32)
/* Output an odd numbered line followed by blanks for the last line if there
/* were an odd number of members listed.
SAY | odd | even |

-TOTAL
padding = COPIES(' ',28)
SAY '+'COPIES("-",67)+'
member_number = SUBSTR(member_number,1,4)
SAY |          TOTAL MEMBERS LISTED WAS member_number padding |
SAY '+'COPIES("-",67)+'
MPR TOP;
MPR DELETE line_count;                /* Remove the original output lines.
MPR NOPRINT SET OUTPUT-EDIT OFF
MPR NOPRINT SET EXEC-WRITE ON;
MPR NOPRINT PULL;

```

FASTQUIT (PROCL-03)

```
MPXX
/* FASTQUIT executive routine
/* This executive routine allows you to immediately XQUIT from any number
/* of buffers and outputs the buffer types exited. If you are initially in
/* Command Mode then there is no need to XQUIT. This routine takes no
/* parameters.

If &MODE EQ C THEN EXIT
loop = 0                                /* Loop counter

-LOOP
loop = loop+1
MPR XQUIT;                             /* XQUIT from this mode.
IF &MODE ^= C THEN GOTO LOOP           /* Loop until in Command Mode.
MPR CLEAR;
LITERAL #
SAY ***> You just quit from #loop# buffers.#
EXIT 0
```

Quick Sign On (PROCL-04)

```
MPXX
/* This executive routine opens a repository. Its parameters are
/* repository-name, authority and (optionally) status-name.
LITERAL #
IF &P0 EQ "" THEN DO
    SAY #NO REPOSITORY NAME SPECIFIED#
    EXIT
END
IF &P1 EQ "" THEN DO
    SAY #NO USER NAME SPECIFIED#
    EXIT
END
MPR NOPRINT SWITCH OFF MESSAGE LEVEL 1;
MPR DICTIONARY &P0 UDATE;
MPR AUTHORITY &P1;
IF &P2 ^= ' ' THEN DO
    MPR STATUS &P2;
END
MPR NOPRINT SWITCH ON MESSAGE LEVEL 1
SAY #REPOSITORY# &DICT #OPEN FOR# &USER #IN STATUS# &STAT
EXIT
```

Decimal Conversion (PROCL-05)

```

MPXX
/* DECIMAL CONVERSION EXECUTIVE ROUTINE
/* This executive routine calculates the decimal for of a fraction n / m
/* passed as three input parameters.
/* (n * m must be less than 1,000,000,000)
/*
LITERAL #
/* Check that the parameters are numbers separated by '/'
IF TYPE(&P0) = #n# AND &P1 = #/# -
  AND TYPE(&P2) = #N# -
  AND &P3 = '' -
  THEN GOTO VALID
SAY #Input is not in the form 'number / number'#
EXIT

-VALID
/* Check that n * m are less than 1,000,000,000
len_dividend = length(&P0)
len_divisor = length(&P2)
IF len_dividend+len_divisor > 9 THEN DO
  SAY #integers too large#
  EXIT 4
END
val(1) = 9-len_dividend
val(4) = val(1)+1          /* Save multiplying power
integer_nums = 1

-loop
integer_nums = 10*integer_nums
val(1) = val(1)-1
IF val(1) > 0 THEN GOTO loop
/* integer_nums is now 10 to power val(1)-1

/* Multiply numerator by integer_nums
val(2) = integer_nums*&P0
val(3) = val(2)/&P2          /* Divide by denominator
/* Divide by integer_nums to get integer part
quotient = val(3)/integer_nums
/* Subtract integer_nums to get decimal part
decimal = integer_nums*quotient+val(3)
decimal = .||decimal          /* Add decimal point

-loop2
decimal_size = length(decimal) /* Final length of decimal part
IF decimal_size < val(4) THEN DO
  /* Add leading zeros until correct number
  decimal = .0||substr(decimal,2,)
  GOTO loop2
END
/* Concatenate integer and decimal parts
result = quotient||decimal

-loop3
answer_size \ LENGTH(result)
last_digit \ answer_size-1
IF substr(result,last_digit,1) EQ "-" THEN GOTO FINE
IF substr(result,answer_size,1) \= 0 THEN GOTO FINE
result = substr(result,1,last_digit)
GOTO loop3

-FINE
SAY '&P0# / #&P2 = result'
EXIT 0

```

All Occurrences (PROCL-06)

```
MPXX
/* This executive routine scans through the current buffer from top to
/* bottom and displays in a lookaside buffer lines containing a specified
/* string together with the line number.
/* Parameter &P0 is the string to be searched for. Parameter strings are
/* upper-cased so this routine cannot be used to find a string containing
/* lowercase characters.
MPR NOPRINT SET;                                /* Ensure that &MSNO is reset
count = 0
if &p0 eq '' then do
    say No argument given
    exit
end
offset = &lino                                /* Get current line number
MPR TOP;                                        /* Search from top of buffer
-test
MPR NOPRINT LOCATE &p0                          /* Attempt to locate string
if &msno eq 8823 then goto finish              /* String not found
count = count+1
say &lino &curl
goto test
-finish
MPR TOP;
MPR NEXT offset;
if count eq 0 then say 'nothing found'
exit 0
```

Overlay (PROCL-07)

```

MPXX
/* This executive routine overlays lines in the current edit/update buffer.
/* The parameters are: string position number-of-occurrences. If
/* number-of-occurrences = '*' then overlaying is done on every line in
/* the buffer. position and number-of-occurrences both default to 1.
literal #
string = &p0
position = &p1
count = &p2
offset = &lino
if position eq '' then position = 1
if count eq '' then count = 1
if type(position) != N then goto error
if type(count) != N and count != '*' then goto error
if &curl eq '*** TOP OF DATA***' then NEXT;
/* Allow for 'do to end'
if count eq '*' then count = &linc+1-&lino
loop_counter = 0                                /* Loop control
    newline = overlay(string,&curl,position)
    NOPRINT CHANGE \&curl ||newline||\ 1 1;
    loop_counter = loop_counter+1

-loop
if loop_counter ne count then do
    NEXT;
    newline = overlay(string,&curl,position)
    NOPRINT CHANGE \&curl ||newline||\ 1 1;
    loop_counter = loop_counter+1
goto loop
end
TOP;
if offset eq 0 then exit
    NEXT offset;
exit

-error

say 'The parameters are string position number-of-occurrences' exit

```

PF Key Settings (PROCL-08)

```
MPXX
/* This executive routine displays your PFkey settings. It takes no
/* parameters.
MPR NOPRINT PUSH;
LITERAL #
/* Allow the output from QUERY PF to be accessed by this executive routine
MPR NOPRINT SET OUTPUT-EDIT ON;
MPR NOPRINT SET EXEC-WRITE OFF;
MPR NOPRINT SET BLANK-LINE-DISPLAY ON;
MPR QUERY PF;
out_lines = &LINC /* Save number of lines of output
MPR DOWN 1; /* Check whether PF keys defined
IF SUBSTR(&CURL,1,2) EQ NO THEN GOTO ENDIT
MPR UP 1;
line_number = 0
-START
MPR DOWN 1;
PF_number = SUBSTR(&CURL,3,2) /* Get PF key number

-UNDEFINED_KEY_LOOP
line_number = line_number+1
IF line_number GT 24 THEN GOTO DISPLAY /* Check for last line
IF PF_number GT line_number THEN DO /* Check for undefined key
PF(line_number) = #PF#line_number# ** UNDEFINED #
IF line_number LT 10 THEN DO /* Insert zero if value < 10.
PF(line_number) = INSERT(0,PF(line_number),2,1)
END
GOTO UNDEFINED_KEY_LOOP
END
PF(PF_number) = SUBSTR(&CURL,1,33) /* Save PF key number and definition
IF &LINC EQ &LINO THEN DO /* Check for undefined PF keys at
PF_number = 25 /* end of output
GOTO UNDEFINED_KEY_LOOP
END
GOTO START

-DISPLAY
/* Generated header information
SAY
SAY #CURRENT CMR PF KEY SETTINGS - USER# &USER# :#
SAY # #COPIES(##,45)
&L51 = ##
&L52 = COPIES (##,70)
&L53 = &L51&L52&L51
SAY &L53
PF_num = 0
-LINELOOP
PF_num = PF_NUM+1
&L31 = #: #
&L34 = PF(PF_num)||&L31||SUBSTR(PF(PF_NUM+12),1,32)
&L35 = INSERT(, &L34,,3)
SAY LEFT(&L35,72,|) /* Output text in a box.
IF PF_num LT 12 THEN GOTO LINELOOP /* Test for last line.
SAY &L53
SAY
MPR TOP;
MPR DELETE out_lines; /* Delete original QUERY PF output.

-ENDIT
MPR NOPRINT SET BLANK-LINE-DISPLAY OFF;
MPR NOPRINT SET OUTPUT-EDIT OFF;
MPR NOPRINT SET EXEC-WRITE ON;
EXIT
MPR NOPRINT PULL;
```


Compound Interest (PROCL-09)

```

MPXX
/* This executive routine provides the compound interest for a period of one
/* to five years from an initial sum in the range 100 to 100,000. Output is
/* given for annual interest rates of -10 percent to 20 percent.
/* There is one parameter: the amount of money.
/* Interest is paid on an annual basis.
MPR NOPRINT PUSH
MPR NOPRINT SET BLANK-LINE-DISPLAY ON;
literal #
null = ''                                /* Define null array
/* Store parameters in the array initial
arrayname = initial                      /* arrayhi must operate on an expression
parse arg initial ()                     /* that can be substituted when evaluated
/* Validate the input parameters.
/* There should be just one integer parameter.
if arrayhi(arrayname) gt 1 then goto errex1
if arrayhi(arrayname) it 1 then goto errex2
if type(initial) ne 'N' then goto errex2
if initial > 100000 or initial < 100 then goto errex3
b = 90
/* Output header information.
say 'Inflation/interest calculation.'
say 'SUM      INTEREST      1yr      2yr      3yr      4yr      5yr'
-loop
if b > 121 then goto wayout
c(1) = initial*b/100
c(2) = c(1)*b/100
c(3) = c(2)*b/100
c(4) = c(3)*b/100
c(5) = c(4)*b/100
h = right(initial,6,'')
interest = b -100
interest = right(interest,3,'')
/* Right justify compounded sums for output.
c(1) = right(c(1),6,'')
c(2) = right(c(2),6,'')
c(3) = right(c(3),6,'')
c(4) = right(c(4),6,'')
c(5) = right(c(5),6,'')
/* Output new compounded values.
say h interest c(1) c(2) c(3) c(4) c(5)
/* Specify intervals between interest rates.
If b > 114 then b = b+4
If b > 104 then b = b+1
If b < 101 then b = b+5
c() = null
say
goto loop                                /* Loop for next interest rate.
-wayout
exit
/* Output error messages
-errex1
say #Only one parameter please.#
exit 4
-errex2
say #You must enter the initial sum as a parameter.#
exit 4
-errex3
say #The initial sum must be an integer between 100 and 100000.#
exit 4
MPR NOPRINT PULL;

```

ISPF Read, Write and Edit

```
MPXX
/* This executive routine uses the ISPF command and will therefore only
/* run in environments where ISPF services are available.
/* It allows you to edit a repository member using the ISPF editor.
/* It takes one parameter: the name of the member to be edited.
LITERAL #
LOCAL cmr_name

/* Check if member name passed
IF ARG(, #P#) = 0 THEN DO
    SAY 'NO MEMBER-NAME SPECIFIED'
    EXIT 8
END

/* Allocate browse data set
#ISPF SELECT CMD(ALLOC F(T) DA('ms.mspgb.techd.transfer') SHR)#
IF ZERRC ^= 0 THEN EXIT 8

/* Update member -error message if update fails
SET cmr_name ARG(0, #P#)
MPR NOPRINT UPDATE cmr_name;
IF &CCOD ^= 0 THEN DO
    SAY 'MEMBER 'cmr_name' NOT ON REPOSITORY'
    EXIT 8
END

/* Write buffer to data set
MPR WRITE T;

/* Edit data set
#ISPF EDIT DATASET('ms.mspgb.techd.transfer')#

IF ZERRC ^= 0 THEN MPR NOPRINT XQUIT;

/* File in repository
IF ZERRC EQ 0 THEN DO
    MPR NOPRINT TOP;
    MPR NOPRINT DELETE 32000;
    MPR READ T;
    MPR FILE;
END

/* Free data set and return to caller
#ISPF SELECT CMD(FREE F(T))#;
EXIT ZERRC
```

ISPF Variables

```

MPXX
/* This executive routine uses the ISPF command and so will only run in
/* environments where ISPF services are available. It reads and sets ISPF
/* variables.
LITERAL #

/* Change the ISPF command-line variable and store the change in the profile
#ISPF PUT ZPLACE BOTTOM#
#ISPF VPUT      ZPLACE PROFILE#;

/* Change the ISPF scroll-amount variable and store the change in the profile
#ISPF PUT ZSCED 13#;
ISPF VPUT ZSCED PROFILE#;

/* Use the ISPF editor to show the values have been changed
#ISPF EDIT DATASET('ms.mspgb.techd.transfer')#;

/* Copy the value of the command-line variable. cmr_zplace does not already
/* exist so ISPF creates it as a command variable.
#ISPF VGET 2PLACE PROFILE#;
ISPF GET ZPLACE cmr_zplace#;

/* Copy the value of the scroll-amount variable. cmr_zsced does not already
/* exist so ISPF creates it as a command variable.
#ISPF VGET ZSCED PROFILE#;
#ISPF GET ZSCED cmr_zsced#;

/* Display the retrieved values
SAY '+=====+'
SAY '|  VARIABLES RETRIEVED FROM ISPF                                |'
SAY '+-----+'
SAY '|  COMMAND LINE      |  ZPLACE      | 'LEFT(cmr_zplace,21)' |'
SAY '+-----+'
SAY '|  SCROLL AMOUNT     |  ZSCED       | 'LEFT(cmr_zsced,20)' |'
SAY '+=====+'

```

7

Executive Commands

This chapter contains specifications, in alphabetical order, of all executive commands.

ARRAYGEN	64
ARRAYGEN Syntax	65
ARRAYSORT	65
ARRAYSORT Syntax	66
BUILD	66
Building a KEPT-DATA List from an Array	67
Building an Array from a KEPT-DATA List	68
BUILD Syntax	69
CLOSEF	70
CLOSEF Syntax	71
DACCESS	72
Obtaining Security, Current Status, and History Information	73
Obtaining Full Status Information	76
Suppressing Information	77
Maintaining Variables for Two or More Members	78
Obtaining Condition Information	80
Example	81
DACCESS Syntax	83
DEXPAND	84
Expanding a Member for a Particular Language	85
Using a Specific Form and Version of Any Processed Items	85
Generating Local Names as Variables	86
Giving Specified Alias Names	86
Example	87
Maintaining Variables for Two or More Members	89
DEXPAND Syntax	91
DRELEASE	91
Rules on Releasing Variables	92
DRELEASE Syntax	93
DRETRIEVE	94
Retrieving Repeating Clauses	95
Retrieving Used-By or Reference Information	96
Example 1	99

Example 2	99
Specifying Variable Names.	100
Suppressing Information	101
Accessing DEXPANDED Information	102
Example.	103
Retrieving Unique Key Identifiers	103
Maintaining Variables for Two or More Members.	104
DRETRIEVE Syntax	105
RELINQUISH	106
RELENQUISH Syntax	107
RESERVE	107
RESERVE Syntax	108
SENDF	108
Sending Output to a USER-MEMBER.	109
Sending Output to a Sequential Dataset	110
Sending Output to a Partitioned Dataset	112
SENDF Syntax	112
SREAD	113
SREAD Syntax	114

ARRAYGEN

The ARRAYGENT command sets up an array where each element represents a line of output from a specified Manager Products command.

The ARRAYGEN command creates a command variable and puts into it the output from the specified Manager Products command, each line of output being stored in a separate element of the variable. The output includes Manager Products messages.

If the array already exists, it is erased, then re-declared.

The ARRAYGEN command rejects the following commands:

- Commands that require the Basic Interactive Facility (such as TOP, BOTTOM, LEFT and RIGHT)
- Commands that require the Extended Interactive Facility (such as CHANGE, UPDATE and EDIT).

The instructions:

```
LITERAL #
ARRAYGEN #DATA_LIST# 'MP-AID LIST USER ;' ;
```

put output from the command:

```
MP-AID LIST USER;
```

into the array DATA_LIST. The first non-blank line is in DATA_LIST(1), the second in DATA_LIST(2), and so on.

Normally blank lines are not included in the array. If you want blank lines included, use the ALL keyword, for example:

```
ARRAYGEN #DATA_LIST# 'MP-AID LIST USER ;' ALL ;
```

If no output is written to the specified variable, then use of the ARRAYH1 function returns a value of 0.

ARRAYGEN Syntax

```

>>—ARRAYGEN array-name 'manager-products-command'—————>
>—[ ALL ]———[ ; ]—————><

```

where:

array-name is the name of an array

manager-products-command is a Manager Products command, optionally including a terminator.

ARRAYSORT

The ARRAYSORT command sorts the contents of a command or global array.

The ARRAYSORT command sorts the contents of a source array according to a key which you specify, and writes the sorted version to a target array. You must specify the following:

- The source and target array names. These must conform to the naming rules for Manager Products variables, and may be no longer than 50 characters. The names can be the same, in which case the source array is overwritten with a sorted version.
- Whether each array is a global or command array
- The start position and length of the key you wish to use. If the key field extends beyond the length of an element, the element is padded with spaces.
- Whether to sort in ASCENDING or DESCENDING order. The sort uses alphanumeric values, not numeric.

No default values are supplied, so you must provide all the required parameters in the syntax. All keywords may be truncated to a single character.

Return codes &CCOD are set as follows:

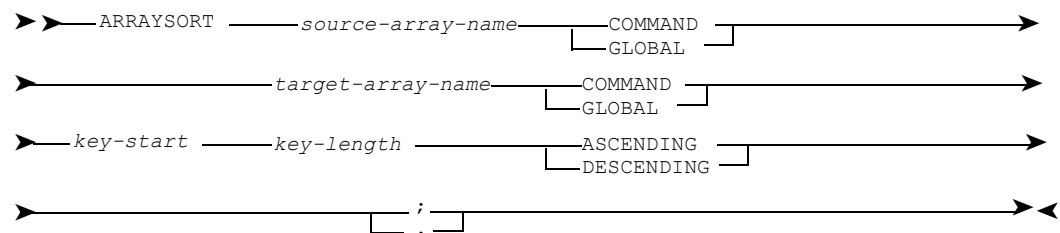
- If the source-array-name does not exist, &CCOD is set to 8
- If the source-array-name is empty, &CCOD is set to 4

For example:

```
ARRAYSORT USERVAR GLOBAL SORTVAR GLOBAL 5 25 DESCENDING ;
```

sorts, in descending order, the contents of global array USERVAR. The results are written to global array SORTVAR. The key field has a start position of 5 and a length of 25.

ARRAYSORT Syntax



where:

key-start must be numeric within the range 1 to 255

key-length must be numeric within the range 1 to 255

BUILD

The BUILD executive command allows the exchange of data between a KEPT-DATA list and an array. You can process the data in whatever form, KEPT-DATA list or array, that is most suitable and then convert back to the original form.

You can:

- Generate a KEPT-DATA list from an array of member key values
- Generate an array of member key values from a KEPT-DATA list

Building a KEPT-DATA List from an Array

Use the BUILD KEPT-DATA command to generate a KEPT-DATA list from an array of member key values.

To generate a named KEPT-DATA list from an array of member key values, enter:

```
BUILD KEPT-DATA IN kept FROM ARRAY #array# KEYS;
```

where:

kept is the name of a KEPT-DATA list

array is the name of an array

is a literal delimiter.

To generate a unnamed KEPT-DATA list from an array of member key values, enter:

```
BUILD KEPT-DATA FROM ARRAY #array# KEYS ;
```

where *array* is the name of an array.

If you specify the ALSO keyword, the BUILD KEPT-DATA command appends members to the specified KEPT-DATA list.

The command generates the KEPT-DATA list from a local array if one exists, otherwise from a command or global array.

The arrays that can be generated from the DRETRIEVE command are shown in the table below. For example, the array ALIAS_KEY is generated by the DRETRIEVE ALL ALIAS-KEYS command.

DRETRIEVE ALL Keyword	Generated Array Name
ALIAS-KEYS	ALIAS_KEY
ATTRIBUTE-KEYS	ATTRIBUTE_KEY
CATALOGUE-KEYS	CATALOGUE_KEY
REFERENCES	REF_KEY
USED-BYS	USED_KEY

You can alter an array of member key values as follows:

- Set an element to zero if you don't want a key value processed by a subsequent BUILD KEPT command
- Set an element to null if you don't want a key value, or any key values with higher element numbers, processed.

Do not alter an array of member key values in any other way or you may generate corrupt KEPT-DATA lists.

Example

This example uses the DACCESS and DRETRIEVE commands to store the references to other members from member FIL1 in array REF_KEY. The BUILD command then generates a KEPT-DATA list.

```
MPXX LITERAL=#  
DACCESS MEMBER FIL1 ;  
DRETRIEVE ALL REFERENCES ;  
BUILD KEPT-DATA IN COLLECTION FROM ARRAY #REF_KEY# KEYS ;
```

Building an Array from a KEPT-DATA List

Use the BUILD ARRAY command to generate an array of member key values from a KEPT-DATA list.

To build an array from a named KEPT-DATA list, enter:

```
BUILD ARRAY array KEYS FROM KEPT IN #kept;
```

where:

array is the name of an array

kept is the name of a KEPT-DATA list

is a literal delimiter.

To build an array from the unnamed KEPT-DATA list, enter:

```
BUILD ARRAY array KEYS FROM KEPT;
```

where *array* is the name of an array.

To append member key values to an existing array of member key values use the ALSO keyword. The key values are added to the array starting at the first null element.

The rules for creating the array or appending to it are as follows:

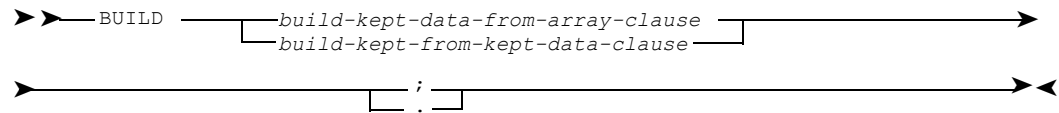
- If ALSO is not specified, all arrays with the specified name are dropped and a local array is created
- If ALSO is specified, a local array with the specified name is created if it does not already exist, and member key values are appended starting at the first empty element.

Example

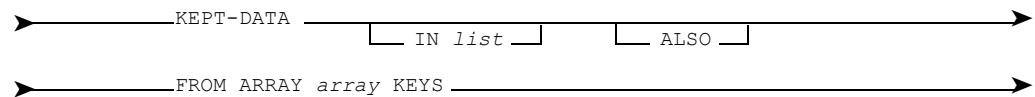
To generate an array DATA_BANK from a KEPT-DATA list ASSORTMENT, enter:

```
BUILD ARRAY DATA_BANK KEYS FROM KEPT IN #ASSORTMENT# ;
```

BUILD Syntax



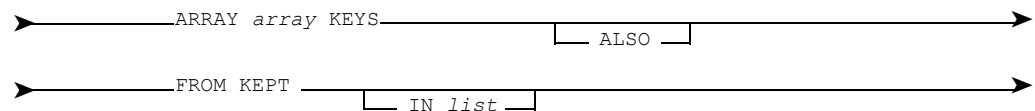
where *build-kept-data-from-array-clause* is:



list is the name of a KEPT-DATA list

array is the name of an array containing key values

build-array-from-kept-data-clause is:



list is as defined above

array is as defined above

CLOSEF

The CLOSEF command closes an output destination opened in a previous SENDF command.

To close an MP-AID USER-MEMBER, enter:

```
CLOSEF member-type user-name;
```

where:

member-type is PUBLIC-USER-MEMBER, PRIVATE-USER-MEMBER, or USER-MEMBER. These keywords are equivalent to each other. They are included only for commonality with the corresponding SENDF command. You therefore cannot use CLOSEF to change a USER-MEMBER from public to private or vice versa.

user-name is the name of the USER-MEMBER.

If a USER-MEMBER destination was specified as NEW or REPLACE in the previous SENDF command, but has not been written to, then the member is not created (empty USER-MEMBERS are not allowed).

To close a sequential dataset, enter:

```
CLOSEF SEQUENTIAL ddname;
```

To close an open member of a partitioned dataset, enter:

```
CLOSEF PARTITIONED ddname MEMBER member-name;
```

where:

ddname is the logical file name used in job control statements to define the external dataset name of the file.

member-name is the name of the member to be written to the partitioned dataset. If this member is not empty, it is stowed onto the dataset, then the dataset is closed. If a member name is not specified, all open (non-empty) members are stowed, then closed.

To close all open members of a partitioned dataset, enter:

```
CLOSEF PARTITIONED ddname;
```

To close the primary/secondary output device (MPOUT) as a destination, enter:

```
CLOSEF PRINT;
```

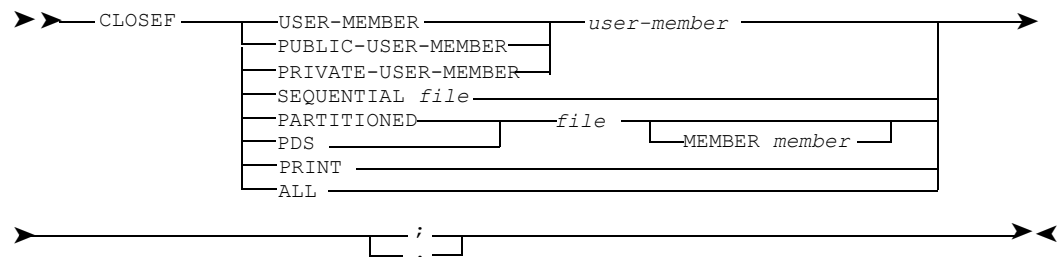
To close all previously-opened destinations, enter:

```
CLOSEF ALL;
```

At the end of the current executive routine (or the highest level executive routine, for nested executive routines), all partitioned datasets or USER-MEMBERS not explicitly closed by a CLOSEF command are automatically closed. Sequential datasets stay open until explicitly closed by a CLOSEF command.

To access the data sent to a destination within the same executive routine run in which the data was output, you must first explicitly close that member.

CLOSEF Syntax



where:

user-name is the name of an MP-AID USER-MEMBER

file is the name of a sequential or partitioned dataset. It is the logical file name (ddname or dtfname) used in job control statements to define the external dataset name (physical file name) of the file

member is the name of a member of a partitioned dataset.

DACCESS

The DACCESS command performs the following functions:

- It reads a specified index-name in the repository
- It performs a security check for protected members to prevent unauthorized users from reading a member's definition
- Subject to the security check, it brings the complete definition of an index-name into an area of virtual storage. Updates to that index-name by other users will not then affect the accessed data.
- It makes non-repeating clauses available for processing (variables occurring in repeating clauses are generated only when a DRETRIEVE command is issued in respect of a DACCESSED member) by generating command variables
- It indicates the occurrence of repeating clauses, by setting up variables which can be used as a control mechanism in the executive routine:
- A COUNT_clause-id variable (referred to as a count variable) representing the total number of occurrences of a repeating clause; and
- An OCC_clause-id variable (referred to as an occurrence variable) to which a subsequent DRETRIEVE command will assign a value depending on the occurrences of the repeating clause. Definition refers either to information that describes the attributes of an index-name and/or to information about relationships of members with catalog classifications, indexed attributes, alias types and names and other members.

At the end of an executive routine any command variables generated for a index-name which has been DACCESSED within that routine are automatically released.

Variables generated for an index-name are nullified if a subsequent DACCESS command is run before the previous index-name is released, unless a cursor number is specified. They will be regenerated when the second DACCESSED member is released.

If CURSOR *c* is specified in a DACCESS command, then the command variables generated are suffixed with a cursor number. This will allow variables from two or more members/index-names to be available at the same time.

To access a member, enter:

```
DACCESS MEMBER name;
```

where *name* is the name of a repository member.

To access a catalog classification record, enter:

```
DACCESS CATALOGUE classification;
```

where *classification* is a catalog classification.

To access an alias record, enter:

```
DACCESS ALIAS alias-name;
```

where *alias-name* is the name of an alias.

To access an indexed clause, enter:

```
DACCESS ATTRIBUTE clause-name;
```

where *clause-name* is the name of a clause.

In the last three cases individual members which use the specified catalog classification, alias or indexed attribute can then be found if the DRETRIEVE USED-BYS/REFERENCES command is used.

Every index-name has a unique key address (the index-name's key). To DACCESS an index-name by its key, enter:

```
DACCESS KEY key;
```

where *key* is a key. You obtain keys by using the DRETRIEVE or BUILD commands.

Obtaining Security, Current Status, and History Information

The command:

```
DACCESS MEMBER member-name;
```

gives you basic information about the member's definition. You can obtain additional information using the WITH clause.

If you enter DACCESS MEMBER *member-name* WITH SITUATION; the following situation variables are generated. Note that the status information generated by WITH SITUATION relates to the currently-visible version of the member; for information about all statuses, see the WITH STATUS-DETAILS option.

Variable Name	Information Held
ACCESS_LEVEL	The access security level
ACCESS_PERMIT	Is access allowed? (YES or NO)
ACCESS_PROTECTED	Does the member have an access level? (YES or NO)
ALTER_LEVEL	The alter security level

Variable Name	Information Held
ALTER_PERMIT	Is alteration allowed? (YES or NO)
ALTER_PROTECTED	Does the member have an alter level? (YES or NO)
EDITION	The number of times the member has been encoded
ENCODE_DATE	The date the member was last encoded
ENCODE_STATUS	The status in which the encoded record or dummy resides
ENCODE_STATUS_CONDITION	The condition of the ENCODE_STATUS status (UPDATE or READ-ONLY)
ENCODE_TIME	The time the member was last encoded
ENCODE_USER	The user who last encoded the member
INSERTION_DATE	The date the member was inserted
INSERTION_TIME	The time the member was inserted
INSERTION_USER	The user who inserted the member
LOCK	Is the member locked? (YES or NO)
LOCK_EXPIRY_DATE	The date the lock expires
LOCK_EXPIRY_TIME	The time the lock expires
LOCKED_BY_USER	The user who locked the member
OWNER_NAME	The owner name
OWNER_PROTECTED	Is the member owned? (YES or NO)
REF_STATUS	The status in which the member's latest reference table resides
REF_STATUS_CONDITION	The condition of the REF_STATUS status (UPDATE or READ-ONLY)
REMOVE_LEVEL	The remove security level
REMOVE_PERMIT	Is removal permitted? (YES or NO)
REMOVE_PROTECTED	Does the member have a removal level? (YES or NO)
SOURCE_CONDITION	The condition of the source record (ENCODED or UNVERIFIED)
SOURCE_STATUS	The status in which the encoded source record resides

Variable Name	Information Held
SOURCE_STATUS_CONDITION	The condition of the SOURCE_STATUS status (UPDATE or READ-ONLY)
UPDATE_DATE	The date the member was last updated
UPDATE_TIME	The time the member was last updated
UPDATE_USER	The user who last updated the member
USED_STATUS	The status in which the member's latest used-by table resides
USED_STATUS_CONDITION	The condition of the USED_STATUS status (UPDATE or READ-ONLY)
UVS_INSERTION_DATE	The insertion date of the unverified source record
UVS_INSERTION_TIME	The insertion time of the unverified source record
UVS_INSERTION_USER	The user who inserted the unverified source record
UVS_STATUS	The status the unverified source record resides in
UVS_STATUS_CONDITION	The condition of the UVS_STATUS status (UPDATE or READ-ONLY)
UVS_UPDATE_DATE	The date the unverified source record was last updated
UVS_UPDATE_TIME	The time the unverified source record was last updated
UVS_UPDATE_USER	The user who last updated the unverified source record

The variables ACCESS_LEVEL, ALTER_LEVEL, and REMOVE_LEVEL are only made available if you are the repository controller.

The variables with prefix UVS_ are only set when the variable SOURCE_CONDITION is equal to UNVERIFIED.

If you intend to subsequently DRETRIEVE a member's used-bys or references, enter:

```
DACCESS MEMBER member-name WITH USED-BYS;
```

Or

```
DACCESS MEMBER member-name WITH REFERENCES;
```

If you specify WITH USED-BYS/REFERENCES, a member's used-bys and references are read into virtual storage at the same time as its definition. They can subsequently be generated as variables by the DRETRIEVE command.

If you do not specify WITH USED-BYS/REFERENCES, the DACCESsed member may be updated later by another user, causing the information generated by a subsequent DRETRIEVE USED-BYS/ REFERENCES to be out of step when compared with the information generated by your DACCESS command.

You can access all used-bys, references and non-syntax attributes, by entering:

```
DACCESS MEMBER member-name WITH SITUATION USED-BYS REFERENCES;
```

Obtaining Full Status Information

You can use the WITH STATUS-DETAILS clause to generate variables containing information about every status in which the member exists. The following status variables are generated for each status in which the member exists.

Variable Name	When Generated
STATUS-MEMBER-TYPE	If the member is encoded or used/references
STATUS-SOURCE-INSERTION-DATE	If source present
STATUS-SOURCE-INSERTION-TIME	If source present
STATUS-SOURCE-INSERTION-USER	If source present
STATUS-SOURCE-UPDATE-DATE	If source has been altered
STATUS-SOURCE-UPDATE-TIME	If source has been altered
STATUS-SOURCE-UPDATE-USER	If source has been altered
STATUS-UVS-INSERTION-DATE	If altered source not same as encoded source
STATUS-UVS-INSERTION-TIME	If altered source not same as encoded source
STATUS-UVS-INSERTION-USER	If altered source not same as encoded source
STATUS-UVS-UPDATE-DATE	If uvs source altered
STATUS-UVS-UPDATE-TIME	If uvs source altered
STATUS-UVS-UPDATE-USER	If uvs source altered
STATUS-ENCODE-DATE	If source encoded
STATUS-ENCODE-TIME	If source encoded
STATUS-ENCODE-USER	If source encoded
STATUS_LOCK_EXPIRY_DATE	The date the status lock expires
STATUS_LOCK_EXPIRY_TIME	The time the status lock expires
STATUS_LOCK_BY_USER	The user who applied the status lock

Variable Name	When Generated
STATUS-REF-DATE	If different references (that is, the type of a referred-to member has changed)
STATUS-REF-TIME	If different references (that is, the type of a referred-to member has changed)
STATUS-REF-USER	If different references (that is, the type of a referred-to member has changed)

Information about the status structure of the dictionary can be obtained by executing the command member MPCMSTAT which will set up the following global variables for each of the named statuses:

```
MPCM_STATUS-LEVEL
MPCM_STATUS-NAME
MPCM_STATUS-CONDITION
MPCM_STATUS-BASE
```

Suppressing Information

You use the SUPPRESS clause to suppress the generation of certain variables (as described below). A DACCESS SUPPRESS command also suppresses the generation of variables from subsequent DRETRIEVE commands.

Using the SUPPRESS clause is more efficient when you are only interested in a few clauses. Generating variables and reading information that you do not need wastes processing time.

To suppress all attributes and prevent the generation of count variables, enter:

```
DACCESS MEMBER member-name SUPPRESS ALL-ATTRIBUTES;
```

The following variables, relating to a member's identity will be the only variables generated:

- Member name and member type
- Member condition (encoded or dummy)
- Base member type.

If SUPPRESS ATTRIBUTES is specified then no variables will be generated for a member's attributes. The variables relating to the member's identity are generated together with count variables, indicating the presence of repeating clauses.

If SUPPRESS COUNTS is specified, count variables will not be generated. If required, a subsequent DRETRIEVE COUNT command can be entered to generate the count variable for a specified clause.

If you have specified any of:

- SUPPRESS ALL-ATTRIBUTES
- SUPPRESS ATTRIBUTES
- SUPPRESS COUNT

then you can use the WITH ATTRIBUTES or WITH ALL-ATTRIBUTES clause in the DRETRIEVE command to generate the variables associated with the keyword specified. In the following example, all variables are suppressed in the DACCESS command, but the variables relating to the clause specified are generated later with a DRETRIEVE command:

```
DACCESS MEMBER member-name SUPPRESS ALL-ATTRIBUTES;  
DRETRIEVE CURRENT EFFECTIVE-DATE WITH ATTRIBUTES;
```

Note: _____

DRETRIEVE WITH COUNTS retrieves count variables for clauses dependent on the clause being retrieved, such as the count variable of an ELSE sub-clause within a CONTAINS clause.

If SUPPRESS DEFINITION is specified access to the definition of the member being DACCESSED is suppressed. Attributes cannot be generated as variables later. Only DRETRIEVE USED-BYS/REFERENCES can be specified in a later command.

Maintaining Variables for Two or More Members

Information from several DACCESSED (or DEXPANDED) members or index-names can be made available for processing at the same time by associating a cursor number with each member or index-name, thereby suffixing the variable names generated for each member or index-name with the relevant cursor number.

This can be done by entering:

```
DACCESS MEMBER member-name CURSOR c;
```

Or

```
DEXPAND MEMBER member-name CURSOR c;
```

All the variables generated are then suffixed by the cursor number, represented by *c*.

c is an integer in the range 1 to 32767.

For example, the command:

```
DACCESS MEMBER EMP-ADDR CURSOR 5;
```

generates variables suffixed by 5 such as:

```
ACCESS_MEMBER_5  
COUNT_DESCRIPTION_5
```

If a second DACCESS takes place with the same cursor number then the variables for the first DACCESSED member with the same cursor number are temporarily nullified until the second DACCESSED member is released. When the member with the same cursor number is released the variables for the previously DACCESSED member are regenerated.

The CURSOR c clause is used to keep the variables of a first and second DACCESSED member (and others if needed) available for processing.

The DRETRIEVE and DRELEASE commands will act on the last DACCESSED member with the same cursor number.

The command:

```
DRETRIEVE FIRST CONTAINS;
```

would retrieve the first occurrence of a CONTAINS clause for the last member DACCESSED with no cursor number specified.

The command:

```
DRETRIEVE FIRST CONTAINS CURSOR 3;
```

would retrieve the first occurrence of a CONTAINS clause for the last member DACCESSED with a cursor number specified as '3'.

If a DRETRIEVE or DRELEASE CURSOR c command does not correspond to a DACCESS command with the same CURSOR c number then an error message is output.

A DRELEASE command with the CURSOR c clause releases the last member DACCESSED with the specified cursor.

Obtaining Condition Information

You use the **CONDITION** keyword to set variables indicating the condition of a repository member, alias, attribute or catalogue. To reveal a member's condition, enter:

```
DACCESS CONDITION MEMBER member-name;
```

If the specified member is present and accessible, the variable **MEMBER_CONDITION** will be set to one of the following values:

Value	Member Condition
ENCODED	Encoded repository member
DUMMY	Dummy member
ALIAS	Alias name
CATALOGUE	Catalogue name
INDEXED ATTRIBUTE	Indexed attribute name

If the specified member is present but not accessible, the variable **MEMBER_CONDITION** will be set to one of the following values:

Value	Member Condition
NOT-VISIBLE	The member is in another status
SOURCE	The member is not encoded
INACCESSIBLE	The user does not have security access to the member

The **DACCESS CONDITION** command is only rejected if the member is not present in the repository or if there is an error in the command syntax.

The **DACCESS CONDITION** command also sets the variable **ACCESS_MEMBER** and, if the user has security access to the member concerned, the variables **BASE_MEMBER_TYPE**, **MEMBER_KEY** and **MEMBER_TYPE**.

If **WITH STATUS-DETAILS** is specified, the status variables are also set. If **WITH SITUATION** is specified, the situation variables are also set.

The member definition variables are not set. For the information returned in these, you should issue a separate **DACCESS MEMBER** command.

You should release the **DACCESSED** member using the standard **DRELEASE MEMBER** command after performing a **DACCESS CONDITION** command.

Example

Consider a member EMP-ADDR whose definition is as follows:

```
ITEM
REPORTED-AS PIC 'X(50) '
HELD-AS CHARACTER 50
ALIAS 'EMPLOYEE-ADDRESS'
      , COBOL 'EMP-ADDR'
      , ASSEMBLER 'EMPADDR'
CATALOG 'EMPLOYEE'
DESCRIPTION 'CURRENT EMPLOYEE PERMANENT HOME ADDRESS'
EFFECTIVE-DATE '29/03/84'
NOTE 'NEEDS REVIEW AFTER 6 MONTHS'
     'AUTHOR DBA'
```

The executive routine:

```
MPXX
DACCESS MEMBER EMP-ADDR;
VLIST COMMAND
```

produces the following output:

```
ACCESS_MEMBER (C)      00001  'EMP-ADDR'
BASE_MEMBER_TYPE (C) 00001  'ITEM'
COUNT_ACCESS_AUTHORITY (C) 00001  0'
COUNT_ADMINISTRATIVE_DATA (C) 00001  '0'
COUNT_ALIAS (C)      00001  '3'
COUNT_ALIAS_KEYS (C) 00001  '3'
COUNT_ATTRIBUTE_KEYS (C) 00001  '0'
COUNT_CATALOGUE (C) 00001  '1'
COUNT_CATALOGUE_KEYS (C) 00001  '1'
COUNT_COMMENT (C)    00001  '0'
COUNT_COPYRIGHT (C) 00001  '0'
COUNT_DESCRIPTION (C) 00001  '1'
COUNT_FORM_DESCRIPTION (C) 00001  '2'
COUNT_FREQUENCY (C) 00001  '0'
COUNT_NOTE (C)      00001  '2'
COUNT_QUERY (C)     00001  '0'
COUNT_REFERENCES (C) 00001  '0'
COUNT_SECURITY_CLASSIFICATION (C) 00001  '0'
COUNT_SEE (C)       00001  '0'
COUNT_UDR1 (C)      00001  '0'
COUNT_UDR2 (C)      00001  '0'
COUNT_UDR3 (C)      00001  '0'
COUNT_UDR4 (C)      00001  '0'
COUNT_UDR5 (C)      00001  '0'
COUNT_UDR6 (C)      00001  '0'
COUNT_UDR7 (C)      00001  '0'
COUNT_UDR8 (C)      00001  '0'
```

```
COUNT_UDR9 (C)      00001  '0'
COUNT_SEE (C)      00001  '0'
COUNT_USED_BYS (C)  00001  '4'
EFFECTIVE_DATE (C)   00001  '1984089'
MEMBER_CONDITION (C)  00001  'ENCODED'
MEMBER_ER_INTEGRITY (C) 00001  'CHECK-OK'
MEMBER_KEY (C)       00001  '332876'
MEMBER_TYPE (C)      00001  'ITEM'
OBSOLETE_DATE (C)     INDEX ENTRY ONLY
OCC_ACCESS_AUTHORITY (C) 00001  '0'
OCC_ADMINISTRATIVE_DATA (C) 00001  '0'
OCC_ALIAS (C)         00001  '0'
OCC_ALIAS_KEYS (C)    00001  '0'
OCC_ATTRIBUTE_KEYS (C) 00001  '0'
OCC_CATALOGUE (C)     00001  '0'
OCC_CATALOGUE_KEYS (C) 00001  '0'
OCC_COMMENT (C)       00001  '0'
OCC_COPYRIGHT (C)     00001  '0'
OCC_DESCRIPTION (C)   00001  '0'
OCC_FORM_DESCRIPTION (C) 00001  '0'
OCC_FREQUENCY (C)     00001  '0'
OCC_NOTE (C)          00001  '0'
OCC_QUERY (C)         00001  '0'
OCC_REFERENCES (C)    00001  '0'
OCC_SECURITY_CLASSIFICATION (C) 00001  '0'
OCC_SEE (C)           00001  '0'
OCC_UDR1 (C)          00001  '0'
OCC_UDR2 (C)          00001  '0'
OCC_UDR3 (C)          00001  '0'
OCC_UDR4 (C)          00001  '0'
OCC_UDR5 (C)          00001  '0'
OCC_UDR6 (C)          00001  '0'
OCC_UDR7 (C)          00001  '0'
OCC_UDR8 (C)          00001  '0'
OCC_UDR9 (C)          00001  '0'
OCC_USED_BYS (C)      00001  '0'
```

Refer to the VLIST directive for details of the format of this output.

The member EMP-ADDR has three ALIAS entries, therefore the COUNT_ALIAS variable has a value of '3'. Repeating clauses such as ALIAS, having a count variable, need to be RETRIEVED in order to access the data contained in the clause. If EMP-ADDR had no aliases then COUNT_ALIAS would be '0'.

The OCC_ variables are set up for each repeating clause. Values are assigned to these when a subsequent DRETRIEVE command retrieves a particular occurrence of a clause.

COUNT_REFERENCES gives the number of members that EMP-ADDR refers to.
COUNT_USED_BYS gives the number of members that refer to EMP-ADDR.



c is an integer in the range 1 to 32767.

DEXPAND

The DEXPAND command generates data structure(s) represented by members. The member types that can be generated from and the rules on following reference paths are the same as for the PRODUCE command.

By processing a specified member's data definition and the members referred to in that definition, DEXPAND generates records representing:

- The hierarchy, group structure and storage offsets or relative start positions of a data structure, and
- The names and definitions of the members constituting it.

The command can be used for:

- Generating variables equivalent to the record layout of a data structure
- Processing members for export to source programming languages
- Producing data descriptions and control statements in the major data base management system languages.

To generate a data structure, enter:

```
DEXPAND MEMBER member-name;
```

To expand a member and to specify a CURSOR number for the variables generated for that member, enter:

```
DEXPAND MEMBER member-name CURSOR c;
```

The records created by the DEXPAND command are made available to the user by the DRETRIEVE EXPAND-RECORD command.

The member DEXPANDED is always the first record generated by the DEXPAND command.

Contained members will be included in the data structure output if:

- They are encoded, and
- They are not protected against access by the user.

For DUMMY members, DEXPAND generates default variables where possible.

The publication *ASG-Manager Products Source Language Generation Facility* discusses record layouts and the PRODUCE command in detail.

Expanding a Member for a Particular Language

To expand a member for a particular language, enter:

```
DEXPAND MEMBER member-name FOR language;
```

The FOR language clause determines how the member is expanded and performs name checks so that all variables produced conform with the language specified. If the FOR language clause is specified the VERIFIED-NAME variable is edited to conform to the language specified. If exporting to that environment, you should use the VERIFIED-NAME as input.

The languages specified can be:

- COBOL
- PL/1
- PL/1F
- BAL.

Using a Specific Form and Version of Any Processed Items

To expand a member using a specific form and version of any ITEM members processed during the expansion, enter:

```
DEXPAND MEMBER member-name USE form VERSION v;
```

form refers to an ITEM's ENTERED-AS, HELD-AS or REPORTED-AS entry, or to its DEFAULTTED-AS entry.

The keywords USE and USING are synonymous. If present, the clause states which form and version of the ITEM members are to be used in the DEXPAND command.

If VERSION *v* is not present in the USE clause, VERSION 1 is the default.

If a form and version stated in the USE or USING clause do not exist for an ITEM member from which generation is taking place, the lowest numbered version of a form selected according to the following order of preference is used:

- DEFAULTTED-AS
- HELD-AS
- ENTERED-AS
- REPORTED-AS.

The first form encountered in this order is taken as the default.

If no USE or USING clause is present, the form and version of ITEM members used are those defined for the containing member. If the member does not state a version the lowest numbered version of the relevant form is assumed. If the member does not state a form, DEFAULTED-AS is assumed.

Generating Local Names as Variables

To expand a member and generate its local names as variables, enter:

```
DEXPAND MEMBER member-name GIVING KNOWN-AS;
```

This clause will, wherever possible, generate variables based on the DEXPANDED members' KNOWN-AS clauses instead of on the member's names or aliases. The VERIFIED-NAME variable will contain the verified KNOWN-AS name.

When both the KNOWN-AS name and the ALIAS name are requested, no ALIAS variable will be generated, if a KNOWN-AS name has been selected.

Giving Specified Alias Names

To expand a member giving specified alias names enter:

```
DEXPAND MEMBER member-name ALIAS alias-type alias-number;
```

Or

```
DEXPAND MEMBER member-name WITH-ALIAS alias-type alias-number;
```

If this clause is present in the command, then names generated are based wherever possible on aliases taken from a member's ALIAS clause. The VERIFIED-NAME variable will contain the verified alias.

If the ALIAS clause specifies an alias-type, each generated alias is based if possible on the specific alias of the specified type in the member's ALIAS clause, unless the KNOWN-AS name is specified.

If an alias number is specified then a general alias with the same number will be returned.

Example

The DEXPAND command generates variables representing the hierarchy, group structure and storage offsets or relative start positions of a data structure, and the names and definitions of the members constituting it.

For example, the following executive routine:

```
MPXX
DEXPAND MEMBER EMP-IDENT;
DRETRIEVE ALL EXPAND-RECORD;
VLIST COMMAND
```

generates these variables:

```
ACCESS_PERMIT      (C)      00001 'YES'
ACCESS_PERMIT      (C)      00002 'YES'
ACCESS_PERMIT      (C)      00003 'YES'
ALIAS_NAME         (C)      INDEX ENTRY ONLY
ALIGNMENT          (C)      00001 'UNALIGNED'
ALIGNMENT          (C)      00002 'UNALIGNED'
ALIGNMENT          (C)      00003 'UNALIGNED'
BASE_MEMBER_TYPE   (C)      00001 'GROUP'
BASE_MEMBER_TYPE   (C)      00002 'ITEM'
BASE_MEMBER_TYPE   (C)      00003 'ITEM'
BIT_OFFSET         (C)      INDEX ENTRY ONLY
COMPRESSED         (C)      INDEX ENTRY ONLY
COUNT_EXPAND_RECORD (C)      00001 '3'
DATA_TYPE          (C)      00002 'CHARACTER'
DATA_TYPE          (C)      00003 'CHARACTER'
DISTINCT_TYPE      (C)      INDEX ENTRY ONLY
EXPANDED_MEMBER    (C)      00001 'EMP-IDENT'
FIELD_LENGTH       (C)      00001 '100'
FIELD_LENGTH       (C)      00002 '50'
FIELD_LENGTH       (C)      00003 '50'
FORM               (C)      00001 'HELD-AS'
FORM               (C)      00002 'HELD-AS'
FORM               (C)      00003 'HELD-AS'
FRACTION_DIGITS    (C)      INDEX ENTRY ONLY
INDEXED            (C)      INDEX ENTRY ONLY
INTEGER_BOUND      (C)      INDEX ENTRY ONLY
JUSTIFICATION      (C)      INDEX ENTRY ONLY
KNOWN_AS_NAME      (C)      INDEX ENTRY ONLY
LENGTH_UNIT        (C)      INDEX ENTRY ONLY
LEVEL_NUMBER       (C)      00001 '1'
LEVEL_NUMBER       (C)      00002 '2'
LEVEL_NUMBER       (C)      00003 '2'
MEMBER_CONDITION   (C)      00001 'ENCODED'
MEMBER_CONDITION   (C)      00002 'ENCODED'
MEMBER_CONDITION   (C)      00003 'ENCODED'
MEMBER_NAME        (C)      00001 'EMP-IDENT'
```

```
MEMBER_NAME      (C)      00002 'EMP-NAME '  
MEMBER_NAME      (C)      00003 'EMP-ADDR '  
MEMBER_KEY       (C)      00001 '1171864 '  
MEMBER_KEY       (C)      00002 '68512 '  
MEMBER_KEY       (C)      00003 '332876 '  
MEMBER_TYPE      (C)      00001 'GROUP '  
MEMBER_TYPE      (C)      00002 'ITEM '  
MEMBER_TYPE      (C)      00003 'ITEM '  
MINIMUM_LENGTH   (C)      00001 '100 '  
MINIMUM_LENGTH   (C)      00002 '50 '  
MINIMUM_LENGTH   (C)      00003 '50 '  
NAME_BOUND       (C)      INDEX ENTRY ONLY  
NAME_BOUND_VALUE (C)      INDEX ENTRY ONLY  
OCC_EXPAND_RECORD (C)      00001 '3 '  
OFFSET           (C)      00001 '0 '  
OFFSET           (C)      00002 '0 '  
OFFSET           (C)      00003 '50 '  
PICTURE          (C)      INDEX ENTRY ONLY  
REDEFINED        (C)      INDEX ENTRY ONLY  
REDEFINED_NAME   (C)      INDEX ENTRY ONLY  
REDEFINED_RECORD (C)      INDEX ENTRY ONLY  
REDEFINES_LENGTH (C)      INDEX ENTRY ONLY  
REQ_FORM_VERSION (C)      00001 'YES '  
REQ_FORM_VERSION (C)      00002 'YES '  
REQ_FORM_VERSION (C)      00003 'YES '  
ROUNDED_TRUNCATED (C)      INDEX ENTRY ONLY  
SIGN             (C)      INDEX ENTRY ONLY  
SIGN_POSITION    (C)      INDEX ENTRY ONLY  
SIGN_SEPARATION  (C)      INDEX ENTRY ONLY  
TOTAL_DIGITS     (C)      00002 '50 '  
TOTAL_DIGITS     (C)      00003 '50 '  
USAGE            (C)      INDEX ENTRY ONLY  
VARIABLE_REDEFINES_LENGTH (C)      INDEX ENTRY ONLY  
VERIFIED_NAME    (C)      00001 'EMP-IDENT '  
VERIFIED_NAME    (C)      00002 'EMP-NAME '  
VERIFIED_NAME    (C)      00003 'EMP-ADDR '  
VERSION          (C)      00002 '1 '  
VERSION          (C)      00003 '1 '
```

Refer to the VLIST directive for details of the format of this output.

If a DEXPANDED member is not an ITEM then fields such as VERSION, FRACTION_DIGITS, DATA_TYPE, BIT_OFFSET, JUSTIFICATION, PICTURE, etc. are set to null.

MEMBER_CONDITION contains information as to whether a member is encoded or a dummy.

MEMBER_NAME is the DEXPANDED member's name.

VERIFIED_NAME is the edited and verified name of the DEXPANDED member's name, or the ALIAS or KNOWN-AS name if requested. If a language is specified, the VERIFIED-NAME will conform to the naming conventions used in the named language and should be used for output to that environment.

OFFSET represents the decimal offset of the storage fields computed for the members constituting a storage block or record.

LEVEL_NUMBER is the hierarchical level of the field. Level numbers commence at 0 for CONVENTIONAL-FILES and '1' for GROUP and ITEM members and are incremented by one for successively lower levels.

FIELD_LENGTH is the computed length of the storage field in bytes.

DATA_TYPE is the form-description of a member, if the member is an ITEM.

ALIGNMENT shows whether the ITEM is aligned in storage and on what word boundary.

In addition, both the above variables and the following are generated for contained members. If access is not authorized then the fields for any member a user cannot access are set to null.

REDEFINED is set to 'REDEFINED' if the contained member is followed by an ELSE clause.

REDEFINED_NAME is set to the member name of the first redefined member.

REDEFINED_RECORD is set to the array number of the first redefined member.

REDEFINES_LENGTH is set to the maximum length of all the redefining members.

Maintaining Variables for Two or More Members

Information from several DACCESSED (or DEXPANDED) members or index-names can be made available for processing at the same time by associating a cursor number with each member or index-name, thereby suffixing the variable names generated for each member or index-name with the relevant cursor number.

This can be done by entering:

```
DACCESS MEMBER member-name CURSOR c;
```

Or

```
DEXPAND MEMBER member-name CURSOR c;
```

All the variables generated are then suffixed by the cursor number, represented by *c*.

c is an integer in the range 1 to 32767.

For example, the command:

```
DACCESS MEMBER EMP-ADDR CURSOR 5;
```

generates variables suffixed by 5 such as:

```
ACCESS_MEMBER_5  
COUNT_DESCRIPTION_5
```

If a second DACCESS takes place with the same cursor number then the variables for the first DACCESSED member with the same cursor number are temporarily nullified until the second DACCESSED member is released. When the member with the same cursor number is released the variables for the previously DACCESSED member are regenerated.

The CURSOR *c* clause is used to keep the variables of a first and second DACCESSED member (and others if needed) available for processing.

The DRETRIEVE and DRELEASE commands will act on the last DACCESSED member with the same cursor number.

The command:

```
DRETRIEVE FIRST CONTAINS;
```

retrieves the first occurrence of a CONTAINS clause for the last member DACCESSED with no cursor number specified.

The command:

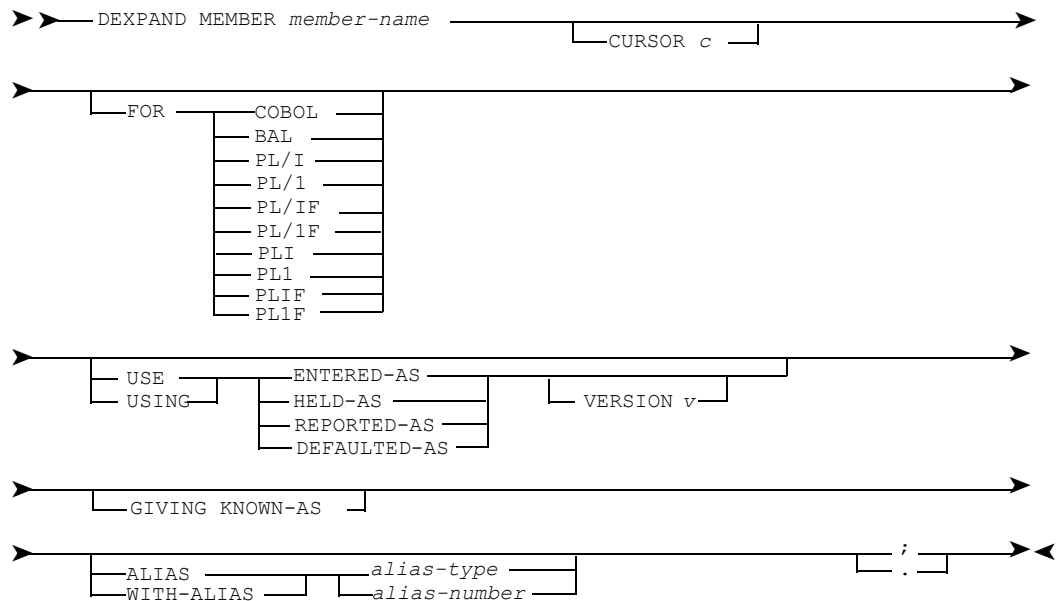
```
DRETRIEVE FIRST CONTAINS CURSOR 3;
```

retrieves the first occurrence of a CONTAINS clause for the last member DACCESSED with a cursor number specified as '3'.

If a DRETRIEVE or DRELEASE CURSOR *c* command does not correspond to a DACCESS command with the same CURSOR *c* number then an error message is output.

A DRELEASE command with the CURSOR *c* clause releases the last member DACCESSED with the specified cursor.

DEXPAND Syntax



where:

c is an integer in the range 1 to 32767

v is an unsigned integer specifying a number from 1 to 15

alias-type is a keyword from the alias-type keyword list of the repository

alias-number can be any number from 1 to the maximum number of general aliases allowed in your repository.

DRELEASE

The DRELEASE command releases variables generated by a DACCESS or DEXPAND command.

To releases a DACCESSED member and nullify the variables generated for it enter:

```
DRELEASE MEMBER member-name;
```

To release a DACCESSED catalog classification, alias or indexed attribute respectively, enter:

```
DRELEASE CATALOGUE catalogue-name;
```

or

```
DRELEASE ALIAS alias-name;
```

or

```
DRELEASE ATTRIBUTE attribute-name;
```

To release a member which has been DEXPANDED with a cursor number, enter:

```
DRELEASE EXPANDED member-name CURSOR c;
```

To release a member which has been DACCESSED by its key, enter:

```
DRELEASE KEY key;
```

It is also possible to release a member which was DACCESSED by its key by using the DRELEASE MEMBER command.

To release all DACCESSED members and all variables relating to them enter:

```
DRELEASE ALL;
```

To release all DACCESSED members accessed in this executive routine and to reactivate DACCESSED members from a higher level executive routine, enter:

```
DRELEASE LOCAL;
```

Rules on Releasing Variables

When a DRELEASE command is issued the index name specified is released. However, if a previous member has been DACCESSED or DEXPANDED, the variables generated by the most recent DACCESS or DEXPAND command are nullified and the originally DACCESSED or DEXPANDED member remains in virtual storage until that too has been DRELEASED.

If CURSOR *c* is specified in the DRELEASE command, the member which is released is the one DACCESSED or DEXPANDED with the same cursor number.

All executive routine variables representing the attributes of a DACCESSED or DEXPANDED member will be set to null, when a DRELEASE command is entered.

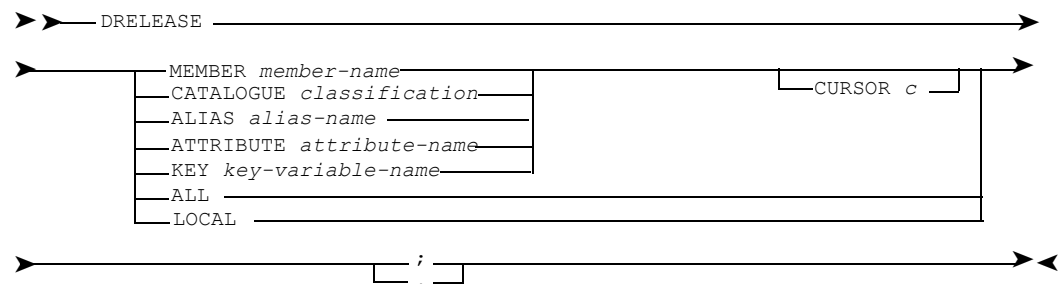
If a DACCESSED or DEXPANDED member superseded a member with the same cursor number which was not DRELEASED then the variables generated by the first member will be re-activated after the second is DRELEASED, and will appear to the user as when previously processed.

If LOCAL is specified then all members or index-names DACCESSEd or DEXPANDEd in this executive routine will be DRELEASEd and any superseded members from a higher level executive routine will be re-activated. This processing also takes place automatically when a user exits an executive routine for a higher level executive routine.

If ALL is specified then all DACCESSEd or DEXPANDEd members are DRELEASEd and all the variables generated by all DACCESS or DEXPAND commands are set to null, regardless of the level of the executive routine.

If the DRELEASE of a member means that the particular member is no longer used by any DACCESS or DEXPAND command then the copy of the member in virtual storage is released. A subsequent DACCESS or DEXPAND command run against that member may pick up a different definition if the member has been updated by other users in the meantime.

DRELEASE Syntax



where:

member-name is the name of a member on the Manager Products repository

class is a catalog classification

alias-name is the name of an alias

attribute-name is the name of an indexed attribute

key-variable-name is the name of a numeric key identifying a member, alias, catalog or indexed attribute on the Manager Products repository

c is an integer in the range 1 to 32767.

DRETRIEVE

The DRETRIEVE command is used:

- To retrieve further information about a previously DACCESSEd member, or
- To retrieve records generated by a DEXPAND command.

The following types of data can be DRETRIEVEd from a DACCESSEd encoded member:

- Text or free form text clauses
- Any repeating clauses in the member's definition
- All used-by or reference information
- Used-by or reference information for a particular relationship type or class
- Catalogue-keys, attribute-keys, and alias-keys
- Non-repeating clauses suppressed in a DACCESS SUPPRESS command.

To see, for a particular member type:

- Which clauses repeat
- The variables associated with each clause

enter:

```
SHOW MEMBER-TYPE FOR MEMBER-TYPE member-type;
```

Records generated by the DEXPAND command can be DRETRIEVEd from a DEXPANDEd encoded member.

If you attempt to DRETRIEVE a clause that is not present in the member an error message is output.

A dummy member can be DACCESSEd, but attempting to DRETRIEVE anything but used-by or reference information is invalid.

When a clause is retrieved, an occurrence variable OCC_clause-id, previously set up by the DACCESS command, is given a value, increasing by one each time an occurrence is DRETRIEVEd.

If a DRETRIEVE clause keyword is the same as the variable name for that clause, you must use literal delimiters. For example:

```
literal #
```

```
DRETRIEVE PREVIOUS #NOTE#;
```

If you do not do this the keyword will be replaced by the variable value during substitution.

If no records are found for the option specified in the DRETRIEVE command an error message is output. ASG recommends that you do a check before issuing the DRETRIEVE. For example:

```
if COUNT_clause-id > 0 then do
    DRETRIEVE FIRST clause-id;
    ...
end
```

Retrieving Repeating Clauses

To retrieve the first occurrence of a repeating clause, enter:

```
DRETRIEVE FIRST clause-id;
```

To access occurrences of a repeating clause one by one in the order they occur, enter:

```
DRETRIEVE NEXT clause-id;
```

If NEXT is requested when no clauses have previously been retrieved then the first occurrence will be returned.

If LAST is specified then the last occurrence of a clause will be returned.

To access in reverse order, use:

```
DRETRIEVE PREVIOUS clause-id;
```

If PREVIOUS is specified when no clauses have previously been DRETRIEVED then the LAST entry will be returned.

If CURRENT is specified the attributes of the current occurrence of the clause will be re-generated. This clause can be used when dealing with data definitions that have common attribute names within unique clauses. For example, to DRETRIEVE the IF clause within a GROUP CONTAINS after DRETRIEVEing the IF from within the GROUP's ELSE clause enter:

```
DRETRIEVE CURRENT CONTAINS;
DRETRIEVE FIRST IF;
```

DRETRIEVE CURRENT is allowed to generate variables suppressed for non-repeating clauses when DACCESS SUPPRESS ATTRIBUTES or SUPPRESS ALL-ATTRIBUTES clause was previously issued.

To access a specific occurrence of a variable in a clause, enter:

```
DRETRIEVE OCCURRENCE n clause-id;
```

where *n* is the occurrence number of a variable in a clause.

If the SUPPRESS COUNTS or SUPPRESS ALL-ATTRIBUTES clause was specified in the DACCESS command (preventing count variables being generated), DRETRIEVE COUNT is used to return the count variable for a specified repeating clause, that is, it sets up the COUNT_*clause-id* variable. For example:

```
DRETRIEVE COUNT NOTE;
```

If the value of the count variable is not zero, the user can then DRETRIEVE the clause.

With the clauses FIRST, NEXT, LAST, PREVIOUS, and OCCURRENCE, all attributes for any clauses associated with any previous occurrence of that clause will be nullified and new control information will be generated for the lower level occurrences associated with the clause.

To access all the occurrences of a clause, enter:

```
DRETRIEVE ALL clause-id;
```

If ALL is specified, the attributes of each occurrence of the clause will be made available in arrays.

Retrieving Used-By or Reference Information

To retrieve all used-by information, information about members that use the DACCESSed member, enter:

```
DRETRIEVE ALL USED-BYS;
```

(This command corresponds to the WHICH MEMBERS USE command.)

To retrieve all reference information, information about members that are used by the DACCESSed member, enter:

```
DRETRIEVE ALL REFERENCES;
```

(This command corresponds to the WHICH MEMBERS CONSTITUTE command.)

Variables giving used-by information have the prefix USED_. Variables giving reference information have the prefix REF_.

There are two types of relationship:

- Entity relationship (ER) relationship types
- Entity association (EA) relationship types

To retrieve reference information for a specified relationship type or relationship class, enter:

```
DRETRIEVE ALL REFERENCES keyword VIA name;
```

where *keyword* is one of these:

ASSOCIATED	Name is the name of an EA relationship type. Only the specified EA information is retrieved.
RELATED	Name is the name of an ER relationship type or relationship class. Only the specified ER information is retrieved.
CONNECTED	Name is the name of a relationship type or relationship class. The specified ER and EA information is retrieved.

If keyword is omitted then the CONNECTED keyword is assumed.

For example, to retrieve reference information for the ER relationship type CALLS, enter:

```
DRETRIEVE ALL REFERENCES RELATED VIA CALLS;
```

To retrieve reference information for the EA relationship type CALLS, enter:

```
DRETRIEVE ALL REFERENCES ASSOCIATED VIA CALLS;
```

To retrieve reference information for the relationship type or relationship class CALLS, enter:

```
DRETRIEVE ALL REFERENCES CONNECTED VIA CALLS;
```

The reference information retrieved can include the following:

- The number of relationships (in COUNT_REFERENCES)
- The member names (in REF_NAME)
- The member types (in REF_MEMBER_TYPE)
- The member key values (in REF_KEY)
- The relationship types (in REF_RELATIONSHIP)
- For ER relationships only, the key values of the relationship members (in REF_RELATIONSHIP_KEY)

- The base member types (in REF_BASE_TYPE)
- The forms of referenced members (in REF_FORM)
- The versions of referenced members (in REF_VERSION)
- The forms of the DACCESSED member (in REF_ACCESS_FORM)
- The versions of the DACCESSED member (in REF_ACCESS_VERSION).

If SUPPRESS IDENTITY is specified then the reference information consists of the following variables only:

- REF_FORM
- REF_BASE_TYPE
- REF_KEY
- REF_RELATIONSHIP
- REF_VERSION.

To retrieve a member's first reference, enter:

```
DRETRIEVE FIRST REFERENCES;
```

Retrieving used-by information is similar to retrieving reference information. You use the USED-BY keyword instead of the REFERENCES keyword, and variable names are prefixed by USED_ instead of by REF_.

If the WITH USED-BYS or WITH REFERENCES clause was specified in the DACCESS command the member records are already in virtual storage, but the variables are only generated after the DRETRIEVE. Otherwise the records are read in when you do the DRETRIEVE.

The referring or referenced members can be accessed with a subsequent DACCESS KEY command.

If you only want the names of EA referenced members it is quicker to DRETRIEVE the clause rather than the references. For example, the command:

```
DRETRIEVE ALL CONTAINS;
```

is faster than:

```
DRETRIEVE ALL REFERENCES ASSOCIATED VIA CONTAINS;
```


Example 1

Suppose the group EMP-IDENT uses the item EMP-ADDR via the EA relationship type CONTAINS. The instructions:

```
DACCESS MEMBER EMP-ADDR;
if COUNT_USED_BY > 0 then do
  DRETRIEVE COUNT USED-BYS ASSOCIATED VIA CONTAINS ;
  if COUNT_USED_BY > 0 then do
    DRETRIEVE FIRST USED-BYS ASSOCIATED VIA CONTAINS ;
    vlist command only USED_
  end
end
end
```

give the following output:

```
USED_ACCESS_FORM      (C)      00001 'HELD-AS'
USED_ACCESS_VERSION   (C)      00001 '1'
USED_BASE_TYPE        (C)      00001 'GROUP'
USED_FORM             (C)      00001 'HELD-AS'
USED_KEY              (C)      00001 '6728'
USED_MEMBER_TYPE      (C)      00001 'GROUP'
USED_NAME             (C)      00001 'EMP-IDENT'
USED_RELATIONSHIP     (C)      00001 'CONTAINS'
USED_VERSION          (C)      INDEX ENTRY ONLY
```

Example 2

Suppose the program CM00 references the modules M0 and M1 via the ER relationship type PROGRAM-CONTAINS-MODULE. The instructions:

```
DACCESS MEMBER CM00;
if COUNT_REFERENCES > 0 then do
  DRETRIEVE COUNT REFERENCES RELATED VIA -
  PROGRAM-CONTAINS-MODULE ;
  if COUNT_REFERENCES > 0 then do
    DRETRIEVE ALL REFERENCES RELATED VIA -
    PROGRAM-CONTAINS-MODULE ;
    vlist command only REF_
  end
end
end
```

give the following output:

```
REF_ACCESS_FORM (C)                INDEX ENTRY ONLY
REF_ACCESS_VERSION (C)              INDEX ENTRY ONLY
REF_BASE_TYPE (C)    00001    'MODULE'
REF_BASE_TYPE (C)    00002    'MODULE'
REF_FORM (C)                INDEX ENTRY ONLY
REF_KEY (C)    00001    '5388'
REF_KEY (C)    00002    '10660'
REF_MEMBER_TYPE (C)    00001    'MOD'
REF_MEMBER_TYPE (C)    00002    'MOD'
REF_NAME (C)    00001    'M0'
REF_NAME (C)    00002    'M1'
REF_RELATIONSHIP (C)    00001    'PROGRAM-CONTAINS-MODULE'
REF_RELATIONSHIP (C)    00002    'PROGRAM-CONTAINS-MODULE'
REF_RELATIONSHIP_KEY (C)    00001    '61518'
REF_RELATIONSHIP_KEY (C)    00002    '52467'
REF_VERSION (C)                INDEX ENTRY ONLY
```

Specifying Variable Names

The AS clause is only allowed for the retrieval of attributes or text or free-form text. Any DRETRIEVED attribute or text string will be handed to the executive language software identified by a variable name specified in the AS clause instead of a variable name derived from the clause's identifier keyword. For example:

```
DRETRIEVE ALL DESCRIPTION AS TEXT;
```

If AS is specified, CURSOR c will have no effect on the name specified in the AS clause. However it must be entered to identify the accessed member.

The AS clause will allow common routines to be written for the processing of different clauses. The variable named in the AS clause will not be automatically removed when any controlling record is removed, or when a member is DRELEASED.

Consider the following example. An executive routine calls a second executive routine, TEXTEXEC, to process text clauses. The count variable is checked and then the executive routine is run. The clause identifier keyword is used as a input parameter.

```
...
IF COUNT_NOTE NE 0 THEN DO
TEXTEXEC NOTE ;
...
```

TEXTEXEC might include the following lines:

```
...
DRETRIEVE COUNT &P0 AS TEXT
-LOOP
DRETRIEVE NEXT &P0 AS TEXT;
WRITEF TEXT
IF OCC_TEXT NE COUNT_TEXT THEN GOTO LOOP
EXIT
```

Suppressing Information

The SUPPRESS clause reduces the processing time for an executive routine if the user is interested in only a few clauses.

If the SUPPRESS COUNTS clause is specified in the DACCESS command, DRETRIEVE COUNT can be used to return the number of occurrences of a repeating clause, that is, to set up the count variable for the clause specified.

DRETRIEVE COUNT is used solely for generating a count variable; it cannot be used with the WITH or SUPPRESS clauses.

Consider the following section of an executive routine:

```
DACCESS MEMBER member-name SUPPRESS ATTRIBUTES;
IF COUNT_DESCRIPTION EQ 0 THEN GOTO NEXTMEMBER
DRETRIEVE ALL DESCRIPTION WITH ATTRIBUTES;
```

In the above example, the executive processor will not spend CPU time generating variables that are not needed by the user. The count variable is tested and if it indicates that there is at least one occurrence of a DESCRIPTION clause the executive routine can continue.

If a DACCESS SUPPRESS command is issued, the suppressed variables can be generated for the current DRETRIEVE command if the WITH clause is specified. For example,

```
...
DACCESS MEMBER member-name SUPPRESS ALL-ATTRIBUTES;
DRETRIEVE COUNT clause-id;
IF COUNT_clause-id NE 0 THEN DO
DRETRIEVE ALL clause-id WITH ATTRIBUTES;
...
```

This executive routine suppresses all variables apart from a member's name and type, its "condition" and base type. To retrieve a repeating clause's count variable a DRETRIEVE COUNT command must then be issued. After checking to see that there is at least one occurrence, the required clause can be retrieved with its attributes.

If DRETRIEVE WITH ALL-ATTRIBUTES is specified the attributes and count variables of the clause being processed (suppressed in a previous DACCESS SUPPRESS ALL-ATTRIBUTES) are generated during processing of the current DRETRIEVE command.

DRETRIEVE WITH COUNTS is used to generate count variables for clauses dependent on the clause being DRETRIEVED.

If DRETRIEVE WITH COUNTS is specified then only count variables dependent on the clause being DRETRIEVED will be generated. The following commands:

```
DACCESS MEMBER member-name SUPPRESS COUNTS;  
DRETRIEVE COUNT CONTAINS;  
IF COUNT_CONTAINS NE 0 THEN DO  
DRETRIEVE ALL CONTAINS WITH COUNTS;
```

will make the following COUNT variables (assuming there are six occurrences of the CONTAINS clause) available:

```
COUNT_CONDITION      (C)      00001 '0'  
COUNT_CONTAINS      (C)      00001 '6'  
COUNT_ELSE          (C)      00001 '0'  
COUNT_INDEXED_BY    (C)      00001 '0'
```

If DRETRIEVE WITH ATTRIBUTES is specified then the attributes of the clause being processed (suppressed in a previous DACCESS SUPPRESS ATTRIBUTES) are generated during processing of the current DRETRIEVE command.

If SUPPRESS IDENTITY is specified then the names of *used-bys* or *references* will be suppressed during retrieval of USED-BYS, REFERENCES or 'KEYS'.

Accessing DEXPANDED Information

To make all the information generated by a DEXPAND command available for processing, enter:

```
DRETRIEVE ALL EXPAND-RECORD;
```

To make the records generated by a DEXPAND command available for processing, one by one in the order they occur, enter:

```
DRETRIEVE NEXT EXPAND-RECORD;
```

Until a DRETRIEVE EXPAND-RECORD command is issued, no records will be available.

Example

A member, called EMP-ADDR, has been DACCESSed.

If the executive command:

```
DRETRIEVE ALL ALIAS;
```

is then issued, the following variables are generated (in addition to those already generated by the DACCESS command):

```
ALIAS_NAME      (C)      00001  'EMPLOYEE-ADDRESS '
ALIAS_NAME      (C)      00002  'EMP-ADDR '
ALIAS_NAME      (C)      00003  'EMPADDR '
ALIAS_TYPE      (C)      00002  'COBOL '
ALIAS_TYPE      (C)      00003  'ASSEMBLER '
```

Note that the occurrence variable of ALIAS generated by the DACCESS command is now given a value:

```
OCC_ALIAS      (C)      00001  '3'
```

The occurrence variable has a value of 3 because all occurrences of the ALIAS clause are available for processing. DRETRIEVE ALL ALIAS was specified. All the aliases in the member's definition have been generated in a variable array. If FIRST had been specified, the value of OCC_ALIAS would be 1, and only the first ALIAS_NAME and ALIAS_TYPE would have appeared above.

The occurrence and count variables can be used to control loops. For example:

```
-LOOP
DRETRIEVE NEXT DESCRIPTION;
IF OCC_DESCRIPTION LT COUNT_DESCRIPTION THEN GOTO LOOP
-NEXT
```

Retrieving Unique Key Identifiers

To retrieve all the unique key identifiers for a member's catalog classifications, enter:

```
DRETRIEVE ALL CATALOGUE-KEYS;
```

To retrieve all the unique key identifiers for a member's aliases, enter:

```
DRETRIEVE ALL ALIAS-KEYS;
```

To retrieve all the unique key identifiers for a member's indexed attributes, enter:

```
DRETRIEVE ALL ATTRIBUTE-KEYS;
```

The key variables that are generated can then be accessed with the DACCESS KEY command if required.

Maintaining Variables for Two or More Members

Information from several DACCESSED (or DEXPANDED) members or index-names can be made available for processing at the same time by associating a cursor number with each member or index-name, thereby suffixing the variable names generated for each member or index-name with the relevant cursor number.

This can be done by entering:

```
DACCESS MEMBER member-name CURSOR c;
```

Or

```
DEXPAND MEMBER member-name CURSOR c;
```

All the variables generated are then suffixed by the cursor number, represented by *c*. *c* is an integer in the range 1 to 32767.

For example, the command:

```
DACCESS MEMBER EMP-ADDR CURSOR 5;
```

generates variables suffixed by 5 such as:

```
ACCESS_MEMBER_5  
COUNT_DESCRIPTION_5
```

If a second DACCESS takes place with the same cursor number then the variables for the first DACCESSED member with the same cursor number are temporarily nullified until the second DACCESSED member is released. When the member with the same cursor number is released the variables for the previously DACCESSED member are regenerated.

The CURSOR *c* clause is used to keep the variables of a first and second DACCESSED member (and others if needed) available for processing.

The DRETRIEVE and DRELEASE commands will act on the last DACCESSED member with the same cursor number.

The command:

```
DRETRIEVE FIRST CONTAINS;
```

retrieves the first occurrence of a CONTAINS clause for the last member DACCESSED with no cursor number specified.

The command:

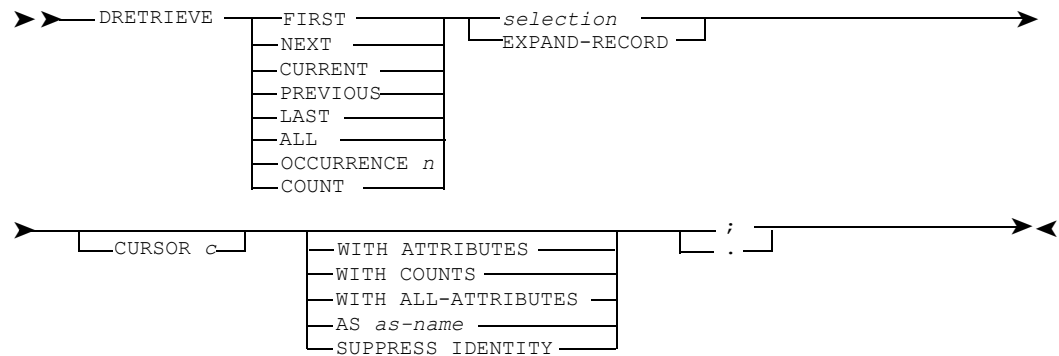
```
DRETRIEVE FIRST CONTAINS CURSOR 3;
```

retrieves the first occurrence of a CONTAINS clause for the last member DACCESSED with a cursor number specified as '3'.

If a DRETRIEVE or DRELEASE CURSOR *c* command does not correspond to a DACCESS command with the same CURSOR *c* number then an error message is output.

A DRELEASE command with the CURSOR *c* clause releases the last member DACCESSED with the specified cursor.

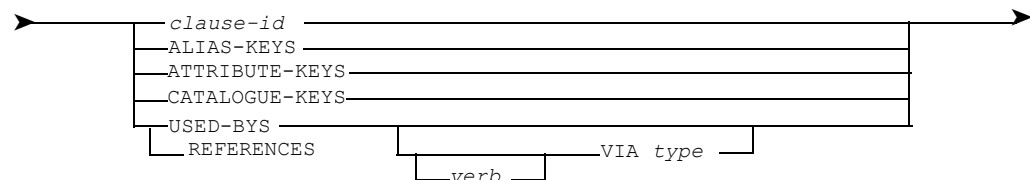
DRETRIEVE Syntax



where:

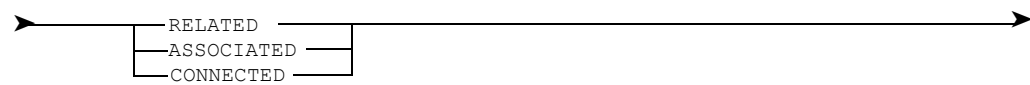
n is an integer in the range 32767

selection is:



clause-id is a clause identifier

verb is:



type is a relationship type or a relationship-type class.

c is an integer in the range 1 to 32767

as-name is a name to replace the generated command variable's name when an attribute, or text, or free form text clause is processed.

RELINQUISH

The RELINQUISH command terminates a Logical Unit of Work (LUW). To terminate an LUW for the repository, enter:

```
RELINQUISH DICTIONARY;
```

To terminate an LUW for the MP-AID, enter:

```
RELINQUISH MP-AID;
```

How transactions carried out as an LUW are processed is determined by keywords specified in the RESERVE and RELINQUISH commands.

If you wish to commit all updates to the repository together when an LUW has finished executing, specify ROLLBACK in the RESERVE command and terminate the LUW by entering:

```
RELINQUISH DICTIONARY COMMIT;
```

In the event of any abnormal termination, no updates are committed and the repository is recovered to the state it was in before the LUW began executing.

You may wish to commit updates only if they produce the result you want, for instance, only if members encode successfully after updating. If they do not, you need not commit the updates, but can instead restore the repository to the state it was in prior to executing the LUW. To do this you need to specify ROLLBACK in the RESERVE command and terminate the LUW by entering:

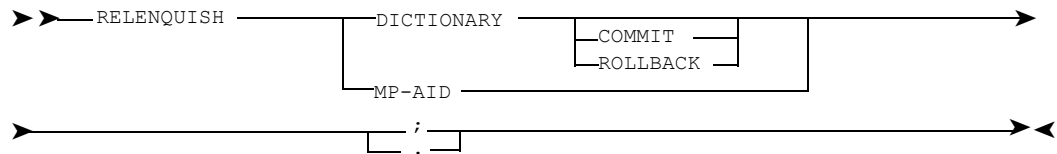
```
RELINQUISH DICTIONARY ROLLBACK;
```

ROLLBACK may be specified only if you specified ROLLBACK in the RESERVE command. If COMMIT is specified (either specifically or by default) in the RESERVE command, you cannot specify ROLLBACK when terminating an LUW.

There are no optional keywords available with the RELINQUISH MP-AID subcommand.

If you do not explicitly terminate an LUW with a RELINQUISH command, the LUW is terminated automatically when the highest level executive routine has finished executing.

RELENGUISH Syntax



RESERVE

The RESERVE command defines the start of a Logical Unit of Work (LUW). An LUW is a group of commands that are treated as one command for the purposes of processing and can be treated as one command for recovery purposes. You can define an LUW for the repository or the MP-AID.

To define the beginning of an LUW for the repository, enter:

```
RESERVE DICTIONARY mode;
```

To define the beginning of an LUW for the MP-AID, enter:

```
RESERVE MP-AID mode;
```

where *mode* is UPDATE if the LUW contains update or a combination of update and interrogation commands or READ-ONLY if the LUW contains interrogation commands only. Updates are not allowed in READ-ONLY LUWs.

Synonyms for these keywords are EXCLUSIVE and SHARED respectively.

To terminate an LUW, use the RELINQUISH command.

If you define a repository LUW in UPDATE/EXCLUSIVE mode, you may choose to commit each update as it completes. Committing an update causes a permanent change to the repository (or MP-AID for an MP-AID LUW). To commit each update as it completes, enter:

```
RESERVE DICTIONARY UPDATE COMMIT;
```

Or

```
RESERVE DICTIONARY UPDATE;
```

If you do not specify a keyword after the mode, the default setting is COMMIT. If any abnormal termination occurs during the processing of a repository update, the repository is automatically recovered to the state it was in before the update began.

Alternatively, you may commit all updates together when the LUW is complete by beginning the LUW:

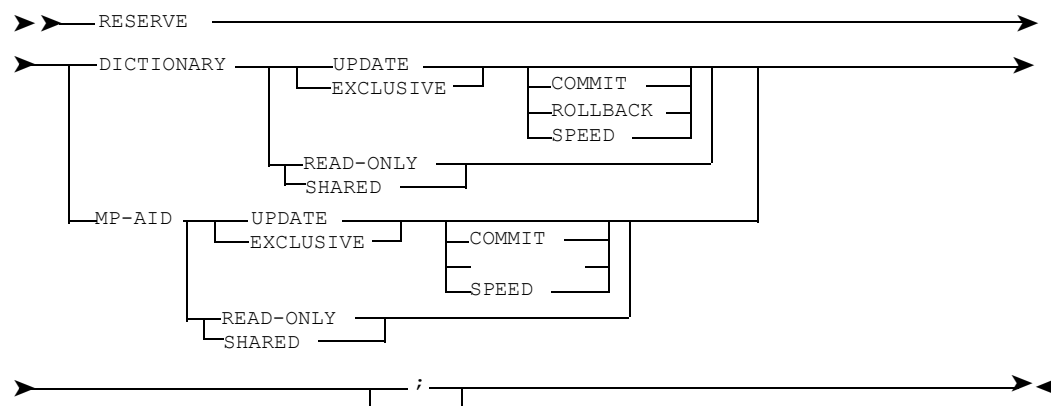
```
RESERVE DICTIONARY UPDATE ROLLBACK;
```

and specifying COMMIT in the RELINQUISH command when you terminate the LUW. If any abnormal termination occurs, the repository is recovered to the state it was in before the LUW began executing.

You cannot specify ROLLBACK when you terminate the LUW if COMMIT is specified (or allocated by default) at the beginning of the LUW, since updates are committed as they complete and cannot therefore be rolled back.

The COMMIT and ROLLBACK keywords are not available in READ-ONLY/SHARED mode or for an MP-AID LUW.

RESERVE Syntax



Note that the SPEED keyword is available for use by Controllers and systems administrators only.

SENDF

The SENDF command opens a destination for subsequent output from one or more WRITEF directives. This destination can be:

- A USER-MEMBER on the MP-AID
- A sequential dataset
- A partitioned dataset

and/or the primary/secondary output device.

To open the primary/secondary output device only (so subsequent WRITEF directives will have the same effect as a SAY directive), enter:

```
SENDF PRINT;
```

If you specify an external file or a USER-MEMBER as a destination, then by default, output is sent to both that destination and the primary/secondary output device.

To send data to that destination only (so that only messages are printed to the primary/secondary output device), enter:

```
SENDF destination NOPRINT;
```

destination specifies an external dataset or USER-MEMBER.

Note:

For the DOS or CICS environments, SENDF is only valid for USER-MEMBERS.

You can use multiple SENDF commands, to open more than one destination, within an executive routine or a nest of executive routines. Output is always sent to the most recently opened destination. To close the most recently opened destination, use the CLOSEF command.

You may want to send data to a previously opened destination. You can use the SENDF command to re-open this destination, without needing to close any other destinations. You cannot alter the characteristics of the destination, as it is already open, so any options specified with a second or subsequent SENDF command are ignored.

Sending Output to a USER-MEMBER

To open a private or public USER-MEMBER on the MP-AID as a destination for output from subsequent WRITEF directives, enter:

```
SENDF member-type member-name options;
```

where:

member-type is a type of USER-MEMBER on the MP-AID:

- USER-MEMBER or PRIVATE-USER-MEMBER, to define the USER-MEMBER as private (only accessible by a user with the currently active Logon Identifier)
- PUBLIC-USER-MEMBER, to define the USER-MEMBER as public (accessible by any Logon Identifier)

When appending or replacing the contents of an existing member the user who created that member can change it from private to public, or the reverse, if the original destination specified by a SENDF command has been closed (by a CLOSEF command or by the termination of the original executive routine). Users with different Logon Identifiers can create private USER-MEMBERS with the same names.

member-name is the name of the USER-MEMBER.

options define how the output is to be filed. These keywords are: NEW, APPEND, or REPLACE.

- NEW: defines the destination USER-MEMBER to be a new member. If the member already exists, the output will not be sent to the USER-MEMBER, the SENDF command will fail and subsequent output from the WRITEF directive will instead be directed to the primary/secondary output device (even if the NOPRINT keyword had also been specified).
- APPEND: subsequent WRITEF directives will append data to the specified USER-MEMBER. If this member does not exist, this has the same effect as NEW.
- REPLACE: subsequent WRITEF directives will be output data to the specified USER-MEMBER, replacing that member if it already exists. If the member does not exist, this has the same effect as NEW.

By default, NEW is assumed for the first WRITEF directive following the SENDF command. For subsequent WRITEF directives to the same destination, the default is APPEND.

Sending Output to a Sequential Dataset

To open a sequential dataset as a destination for subsequent WRITEF directives, enter:

```
SENDER SEQUENTIAL file sequential-options;
```

where:

file is the name of the sequential file. It is the logical file name (ddname or dtfname) used in job control statements to define the external dataset name (physical file name) of the file.

sequential-options define the characteristics of the sequential dataset. Characteristics defined with the SENDF command take precedence over those defined in job control statements.

Characteristics of a new dataset must be defined either in the job control statements for that dataset, or as part of the SENDF command. Characteristics of an existing dataset need not be defined; if they are not, the dataset is replaced with a new dataset, which has the same characteristics.

sequential-options can include the following:

- FORMAT FIXED or FORMAT VARIABLE
- RECORD-SIZE followed by the required logical record size in bytes.
- BLOCK-SIZE followed by the required block size in bytes.

Note: _____

If you specify FORMAT VARIABLE, the maximum length of the user-supplied data is 4 bytes less than the record size you specify.

The permitted values for block and record sizes are:

- For FIXED format:
 - The record length must be between 1 and 32760 bytes
 - The blocksize must be between 16 and 32760 bytes and must be a multiple of the record length
- For VARIABLE format:
 - The record length must be between 5 and 32756 bytes
 - The blocksize must be between 16 and 32760 bytes and must be at least 4 bytes greater than the record length

The following is an example of a SENDF command for output to a FIXED format dataset:

```
SENDF SEQUENTIAL FILE4 FORMAT FIXED RECORD-SIZE 80 BLOCK-SIZE 8000;
```

The following is an example of output to a VARIABLE format dataset:

```
SENDF SEQUENTIAL FILE4 FORMAT VARIABLE RECORD-SIZE 300 BLOCK-SIZE 10000;
```

For fixed length output, short records are right padded as necessary with blanks.

The current executive routine will be terminated if any overlength records are generated by the WRITEF directive.

Sending Output to a Partitioned Dataset

To specify output to a partitioned dataset (FIXED or VARIABLE format), enter:

```
SENDF PARTITIONED file MEMBER member NEW;
```

To specify output to a partitioned dataset, replacing any existing member, enter:

```
SENDF PARTITIONED file MEMBER member REPLACE;
```

where:

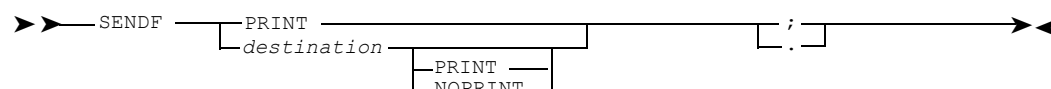
file is the name of the partitioned dataset. It is the logical file name (ddname) used in job control statements to define the external dataset name (physical file name) of the file.

member is the required member name of the partitioned dataset. This dataset must already exist.

Note:

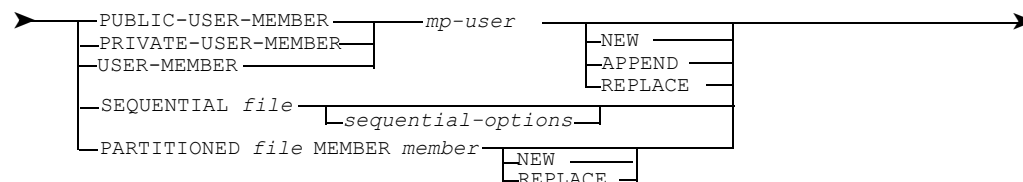
The current executive routine will be terminated if any overlength records are generated by the WRITEF directive.

SENDF Syntax



where:

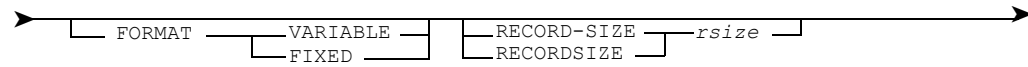
destination is:



mp-user is the name of an MP-AID USER-MEMBER

file is the name of a sequential or partitioned dataset. It is the logical file name (ddname or dtfname) used in job control statements to define the external dataset name (physical file name) of the file.

sequential-options are:



rsize is the record length

bsize is the block size.

member is the name of the member of the partitioned dataset.

SREAD

Input to procedures is normally obtained in-line from an active executive routine which must terminate before input is again read from the Manager Products primary input device (MPIN). Use the SREAD command to read from MPIN instead of from an executive routine.

When you use SREAD, input data from MPIN is read into a Procedures Language command variable named MPCM_SOURCE. This command variable is available to the calling executive upon return from the SREAD command.

Reading from MPIN terminates when a standard Manager Products terminator (semicolon or period) is encountered in position 1 of an input record. Such terminators are not added to the command variable.

In the example below, the primary command REPLACE is reconfigured to invoke the executive routine £REPLACE, using the command:

```
SET PRIMARY-COMMAND REPLACE AS £REPLACE;
```

The executive routine £REPLACE carries out checks on, and modifications to, a user-supplied member definition, as described below. (See the *ASG-Manager Products Systems Administrator's Manual* for information on the SET PRIMARY-COMMAND command.)

£REPLACE uses SREAD to retrieve the source definition of the member to be replaced. If required, the member name and definition is modified. For example:

```

MPXX LITERAL=:
NOPR PUSH;
NOPR SWI OFF MESS NUM 0;
SREAD;
IF SUBSTR(MPCM_SOURCE(1),1,2) EQ 'IT' AND SUBSTR(&P0,1,3) NE 'IT-' -
  THEN CALL PREFIX
ELSE IF SUBSTR(MPCM_SOURCE(1),1,2) EQ 'GR' AND SUBSTR(&P0,1,3) NE -
  'GR-' THEN CALL PREFIX
REPLACE &P0;
DO FOR ARRAYHI(:MPCM_SOURCE:)
MPCM_SOURCE(FDO(DFOR))
END
DESC
MPRE : 'Description line added on :&date: at: &time:':
NOTE
MPRE : 'Note line added on :&date: at: &time:':
;
REPORT &P0;
NOPR PULL;
EXIT 0
-PREFIX
  &P0 = SUBSTR(MPCM_SOURCE(1),1,2):-:&P0
RETURN

```

SREAD Syntax



8

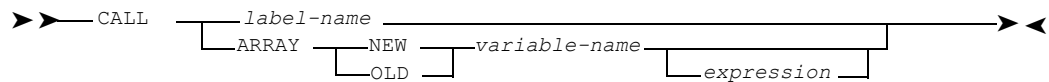
Directives

This chapter contains the specifications of all directives, in alphabetical order.

CALL	116
COMMAND	118
DO	119
DROP	121
EXIT	121
GLOBAL	122
GOTO	123
IF	123
INTERPRET	126
ITERATE	128
LEAVE	129
LITERAL	130
LOCAL	131
MESSAGE	131
MPR	133
MPRE	134
MPXX	134
NOP	135
PARSE	135
PARSEOPTION	137
PROFILE	137
RELEASE	138
RETAIN	140
RETURN	140
SAY	141

SET	141
SIGNAL	142
STACK	143
TRACE	144
TRANSFER	145
VLIST	146
WRITEF	149
WRITEL	151

CALL



where:

label-name is the name of a label

variable-name is the name of a user-defined variable array. The name may have no more than 10 characters.

expression is a parameter to be passed when the user-defined variable array is executed as a procedure

There are two basic types of construct:

- **CALL label-name**, which causes an immediate branch to a label. This is similar to the GOTO directive, except that you return to the most recent active CALL by executing a RETURN directive.
- **CALL ARRAY**, which causes a user-defined variable array to be executed as a procedure.

In both constructs, when a CALL directive is executed, the following settings are stacked:

- The SIGNAL directive settings
- The TRACE directive settings

and active DO loops and instruction blocks are suspended.

CALL label-name Option

When a RETURN directive is executed:

- The most recent copy of the stacked settings is restored
- DO loops and instruction blocks initiated since the most recent active CALL are cancelled
- DO loops and instruction blocks suspended at the most recent active CALL are reactivated.

If there is no active CALL, a RETURN directive causes an error.

The CALL and RETURN directives are useful when you want to call the same piece of code several times, but do not want to put that code into a separate executive routine.

Example of CALL label-name

The executive routine below takes one parameter, an integer, and outputs the factorial of that integer.

```
if arg() ne 1 then exit
if type(&p0) ne N then exit
if &p0 gt 12 then exit
i = &p0
call factorial
say j
exit
-factorial
j = 1
do for i
  j = j*fdo(dfor)
end
return
```

CALL ARRAY Option

Use the keyword NEW if the data which the array currently contains is to be used as the source for building the procedure. Use the keyword OLD if the array has previously been executed as a procedure, and you want this version to be executed without reference to any changes made subsequently to the array. For example:

```
call array old uvar
```

indicates that the user-defined variable array *uvar* should be executed using the currently built procedure, even though changes have since been made to the variable array.

If you pass a parameter to the procedure, it is subject to Limited Substitution.

Array procedures are unaffected by the RETAIN directive and by the SET EXECUTION-RETENTION command. They take the type of the procedure from which they are CALLED as NEW procedures, and can thus execute all instructions which are valid for that type of procedure.

The INTERPRET and LITERAL directives may not be used in array procedures.

CALL ARRAY NEW directives may not be used in array procedures if the named array was used to build a procedure which is still executing.

Procedures built with CALL ARRAY may only subsequently be invoked by the same means. They may not be invoked with the TRANSFER directive or by referring to them as if they were user-defined commands.

Example of CALL ARRAY option

The executive routine below shows how the OLD and NEW keywords can be used to control the directives actually executed.

```
UV1='SAY VERSION 1 '  
CALL ARRAY NEW UV1  
UV1='SAY VERSION 2 '  
CALL ARRAY OLD UV1  
CALL ARRAY NEW UV1
```

When executed, this routine gives the following output:

```
VERSION 1  
VERSION 1  
VERSION 2
```

COMMAND

➤ ➤ ➤ ➤ COMMAND *name* ————— ➤ ➤ ➤ ➤

where *name* is any valid user-defined variable name.

The COMMAND directive declares command variables.

A command variable exists from its declaration until the highest-level executive routine terminates.

If a command variable already exists with the specified name then that declaration has no effect.

If a global variable already exists with the specified name then that variable becomes a command variable.



n is greater than or equal to zero

The DO and END directives group a sequence of instructions together into a block. The block is executed zero or more times.

- DO instructions END

```
IF condition THEN DO
    instruction_1
    instruction_2
    ...
    instruction_n
END
```

- The block is executed once for every non-null element of the array. For example, the instructions:

produce the following output:

119

- DO FOR expression instructions END

The expression must evaluate to a number n that is greater than or equal to zero. The block is executed n times.

- DO UNTIL condition instructions END

The block is executed as long as the condition is false. The block is executed at least once.

- DO WHILE condition instructions END

The block is executed as long as the condition is true. The block can be executed zero times.

In the combined forms:

```
DO array-name() FOR
DO array-name() UNTIL
DO array-name() WHILE
```

the block is executed as many times as there are non-null elements in the array, or until the FOR, UNTIL or WHILE clause stops the block being repeated.

Blocks can be nested. For example:

```
DO FOR expression
  DO WHILE condition
    instruction_1
    instruction_2

    instruction_n
  END
END
```

You should not jump into or out of blocks using the GOTO directive. The only exception to this is if you jump out of all currently active blocks.

You can of course jump within the currently active block.

DROP

```

<<<<
>>>> DROP name

```

where *name* is any valid user-defined variable name.

The DROP directive erases individual user-defined variables. When a variable is dropped, using the DROP directive, the executive routine continues as if the variable had never existed.

If a variable exists as

- A command or global variable, and
- A local variable

then the DROP directive only erases the local variable and the other instance of the variable becomes visible.

EXIT

```

>>>> EXIT
      expression
      EXEC
      SESSION

```

where *expression* is an expression subject to Full Evaluation that evaluates to an integer between 0 and 255 inclusive.

The EXIT directive causes an immediate exit from the current executive routine, and sets return value(s) for the calling executive routine (the executive routine that control returns to).

Note the following uses of the EXIT directive:

- Setting a return code in &CCOD
- Resetting &ECOD or &SCOD when an executive routine has recovered from an error condition.

Refer to ["Return Codes" on page 42](#) for further details of return codes.

There are four versions of the directive:

- EXIT
This is the same as EXIT &ECOD.
- EXIT expression
&CCOD in the calling executive routine is set to the result of evaluating expression.
&ECOD and &SCOD are adjusted if necessary.
- EXIT expression EXEC
This is the same as EXIT expression.
- EXIT expression SESSION
&CCOD, &ECOD and &SCOD in the calling executive routine are all set to the result of evaluating expression.

Example 1

EXIT 4&CCOD is set to 4 and &SCOD and &ECOD are adjusted if necessary.

Example 2

EXIT L20 SESSION

&SCOD, &ECOD and &CCOD are set to the value of the variable L20.

GLOBAL

➤➤— GLOBAL ^{<<<<}*name* —————➤➤

where *name* is any valid user-defined variable name.

The GLOBAL directive declares global variables.

A global variable exists from its declaration to the end of the Manager Products session.

If a global variable of the specified name already exists then the declaration has no effect.

If a command variable of the specified name already exists then that variable becomes a global variable.

GOTO

```

      <<<<
>>>— GOTO label —————>>>

```

where *label* is the name of a label.

The GOTO directive causes control to be passed to the directive following the specified label. There are two types of GOTO:

- Unconditional
- Conditional.

A GOTO directive between the DO and END directives of a DO loop cancels all active DO loops, unless the branch is to a label in the same loop. In that instance, all current DO loops remain in operation. DO instruction blocks are no longer significant in this context, such that a GOTO inside or outside an instruction block may branch to a label outside or inside that block without cancelling the DO loop.

The value returned by the FDO(DLEVEL) function is unreliable if GOTO directives are used to branch into or out of DO instruction blocks.

Examples

Here is an example of an unconditional GOTO:

```

GOTO STORE
-LABEL
.
.
-STORE

```

Here is an example of a conditional GOTO:

```

IF &G3 EQ 'READ' THEN GOTO INPUT
.
.
-INPUT

```

IF

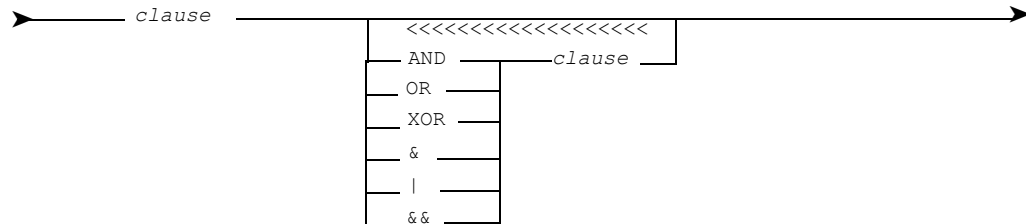
```

>>>— IF condition THEN instruction —————>>>
>>>— [ ELSE instruction ] —————>>>

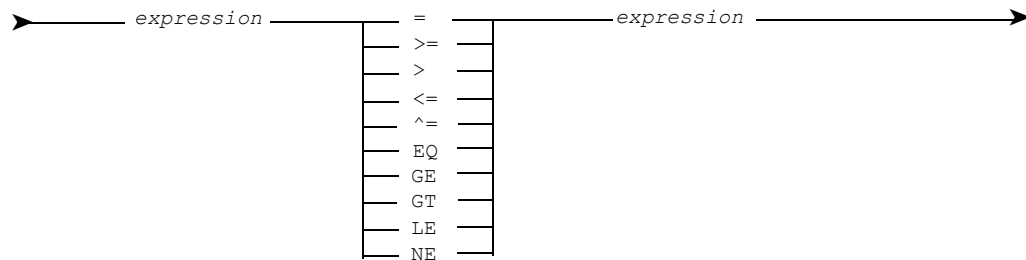
```

where:

condition is:



clause is:



expression is an expression subject to Full Evaluation

instruction is a single command or directive, or a block of instructions

The IF directive is used to conditionally execute instructions.

There are two types of construct.

- IF condition THEN instruction
If the condition is true then instruction is executed, otherwise the construct has no effect.
- IF condition THEN instruction_1 ELSE instruction_2
If the condition is true then instruction_1 is executed, otherwise instruction_2 is executed.

The ELSE keyword must start a new line. For example the instructions:

```
IF i=1 THEN GOTO L1
ELSE GOTO L2
```

are valid, but the instructions:

```
IF i=1 THEN GOTO L1 ELSE GOTO L2
```

are invalid.

An ELSE keyword is always paired with an IF directive. When IF directives are nested each ELSE keyword (if there are any) is paired with the most recent unpaired IF directive. See example 4.

An expression may not include a combination of literal and string delimiters, for example the following is invalid:

```
LITERAL #
IF X = #'# THEN DO
```

Use the following instead:

```
IF X = "'" THEN DO
```

There are two versions of each operator. For example, GE is the same as >= and XOR is the same as &&.

Example 1

Here are some examples of conditions.

The condition:

```
reply EQ 1 AND &P1 EQ 1
```

is true if and only if both reply and &P1 are equal to 1.

The condition:

```
&P0 EQ 'A' XOR &P1 EQ 'B'
```

is true if and only if exactly one of the sub conditions is true.

Example 2

The instruction:

```
IF i=j THEN MPR MYEXEC;
```

causes executive routine MYEXEC to be called if and only if variable i has the same value as variable j.

The valid instructions are:

- Assignment instructions
- Instructions beginning with the directives COMMAND, DROP, GLOBAL, LOCAL, MPR, MPRE, NOP, PROFILE, RELEASE, SAY, VLIST, WRITEF
- Commands.

It is invalid for the instruction to begin with any other directive.

The assignment instructions can be variable assignments:

```
variable_name = expression
```

or they can be array assignments:

```
array_name1() = array_name2
```

Example 1

```
MPXX LITERAL=:
a = :MPR:
b = :MP-AID:
c = :LIST:
d = :TYPE ;:
TYPE = :USER:
INTERPRET a b c d
```

The expressions are evaluated and concatenated to give the instruction:

```
MPR MP-AID LIST TYPE;
```

and the effect is then as if this instruction were inserted into the executive routine in place of the INTERPRET instruction. So the command:

```
MP-AID LIST USER;
```

is executed.

Example 2

```
MPXX LITERAL=:  
A = 3  
INTERPRET :  
GLOBAL TEMP:A
```

The expressions are evaluated and concatenated to give the instruction:

```
GLOBAL TEMP3
```

So a global variable TEMP3 is declared. Thus you can dynamically allocate variable names.

Example 3

Here is routine GET_CLAUSE to print out all occurrences of a given clause in a given member. &p0 is the member name and &p1 is the clause name.

```
mpxx literal=:  
MPR :DACCESS MEMBER: &p0 :SUPPRESS ATTRIBUTES;;  
interpret loop=COUNT_&p1  
do for loop  
    MPR :DRETRIEVE NEXT: &p1;  
    interpret say &p1  
end
```

For example, to print out all occurrences of the NOTE clause in the member DMC28, enter:

```
GET_CLAUSE DMC28 NOTE;
```

ITERATE

➤➤— ITERATE —————➤➤

The ITERATE directive causes an immediate branch to the start of the currently active DO loop. If there is no currently active DO loop, ITERATE is invalid.

Note that a simple DO ... END block does not define a DO loop.

Example

The instructions:

```
do for 4
  i=fdo(dfor)
  if i=3 then iterate
  say i
end
```

produce the following output:

```
1
2
4
```

LEAVE

➤ — LEAVE ————— ➤

The LEAVE directive causes an immediate branch to the first instruction after the currently active DO loop. If there is no currently active DO loop, LEAVE is invalid.

Note that a simple DO ... END block does not define a DO loop.

Example

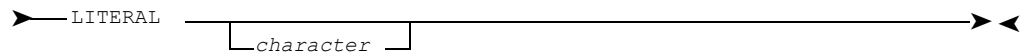
The instructions:

```
do for 4
  i=fdo(dfor)
  if i=3 then leave
  say i
end
```

produce the following output:

```
1
2
```

LITERAL



where *character* is any character except an operator (that is, & + - * / !=) or a string delimiter.

The LITERAL directive defines a literal delimiter. A string in literal delimiters is not evaluated.

Lower case letters are not translated to upper case by the LITERAL directive, so for example the instructions:

```
LITERAL q
LITERAL Q
```

define two literal delimiters, q and Q.

Here is an example of using literal delimiters:

```
LITERAL #
VALUE1 = ETWAS
VALUE2 = #ETWAS#
```

VALUE1 is set to the value of the variable ETWAS and VALUE2 is set to the string ETWAS.

To cancel all literal delimiters, use:

```
LITERAL
```

Consider the expression AAA(9). If AAA is a variable then the expression evaluates to the ninth element of AAA. If AAA is not a variable then it evaluates by convention to the string AAA. To prevent this happening, you can put the expression in literal delimiters.

You are recommended to enclose all literal strings within literal delimiters, thus avoiding any conflict with variable names and saving processing time.

LOCAL

A diagram illustrating the conversion of a local name to a fully qualified domain name. It shows a horizontal line with arrows at both ends. Above the left end of the line are four less-than signs (<<<<). Below the line, the text "LOCAL name" is positioned near the left arrowhead.

where *name* is any valid user-defined variable name.

The `LOCAL` directive declares local variables.

If a local variable of the specified name already exists, the declaration has no effect.

If an assignment is made to an undeclared variable then the variable is automatically declared as a local variable. However, it is good programming practice to explicitly declare all variables.

If a variable exists as:

- A local variable
- A global or command variable

then only the local variable is visible until the local variable is erased using the DROP directive.

MESSAGE

➤ MESSAGE *message-number message-level variable-text* ➡ ➤

where:

message-number is any integer below 65536

message-level is one of the following:

- I or i Informatory message
- W or w Warning message
- E or e Error message
- S or s Serious error message
- C or c Critical error message

variable-text is text for one of the variables in the message.

The MESSAGE directive allows you to output a Manager Products message.

variable-text can be specified 0 to 8 times, each text string containing up to 256 characters.

If *message-number* is not a valid Manager Products message number then the following message is output:

```
DMnnnnnx MESSAGE NUMBER NOT RECOGNIZED
```

where:

nnnnn is the message number

x is the message level

Example 1

The directive:

```
MESSAGE 1292 W FILE1
```

produces the following output:

```
DM01292W FILE1 ENCODING UNSUCCESSFUL *****
```

Note that you may specify a different message level to that normally issued with a particular message by the Manager Products software. In the above example the message is issued as a Warning (W) level message instead of the usual Error (E) level message.

Example 2

The directive:

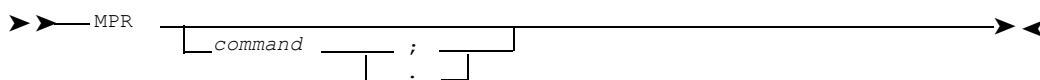
```
MESSAGE 8814 E " 'USER-MEMBER' FILE2
```

produces following output:

```
DM08814E MP-AID USER-MEMBER FILE2 ALREADY PRESENT
```

Note that a null string must be specified as the first item of variable-text in this instance as the standard ASG message format assumes that the first item of variable-text precedes the first word of the message text.

MPR



where *command* is any Manager Products command.

The MPR directive identifies a Manager Products command.

The MPR directive is similar to the MPRE directive, except that Limited Evaluation takes place instead of Full Evaluation.

Normally the procedures language first attempts to interpret an instruction as a directive, then attempts to interpret it as a command.

If you use the MPR directive, the instruction is only interpreted as a command.

As previously stated the expression after the MPR directive is subject to Limited Evaluation. For example:

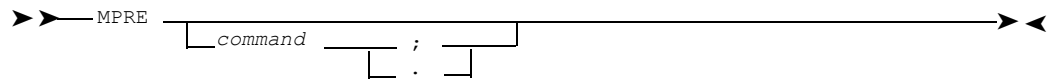
```
MPR SET OUTPUT-EDIT &L9;
```

where *&L9* should contain a value of ON or OFF.

It is essential that all Manager Product commands having the same name as a procedures language directive are prefixed by MPR. Currently these commands are SET, DROP, and TRANSFER.

ASG recommends that you precede all Manager Products commands with MPR as there may in future releases be other directives with the same name as a Manager Product command. This would cause an executive routine which successfully executed with an old release to fail with a new release.

MPRE



where *command* is any Manager Products command.

The MPRE directive identifies a Manager Products command.

The MPRE directive is similar to the MPR directive, except that Full Evaluation takes place instead of Limited Evaluation.

Normally the procedures language first attempts to interpret an instruction as a directive, then attempts to interpret it as a command.

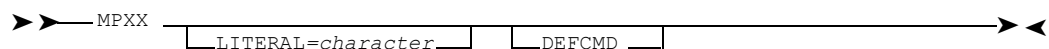
If you use the MPRE directive, the instruction is only interpreted as a command.

As previously stated the expression after the MPRE directive is subject to Full Evaluation. For example:

```
MPRE SENDF USER &LOGO (1,4) &TIME (1,2) &TIME (4,2) ;
```

where substrings of the system variables &LOGO and &TIME are used to build an MP-AID USER-MEMBER name.

MPXX



where *character* is any valid literal delimiter.

The MPXX directive identifies an executive routine.

In an EXECUTIVE-ROUTINE dictionary member, MPXX must be the first directive of the CONTENTS clause. In a USER-MEMBER, MPXX must be the first directive. In either case MPXX must occupy the first four character positions of the first line.

You do not need MPXX in TRANSIENT members.

You can optionally specify a literal delimiter in the MPXX directive. If you do so then any LITERAL directive is invalid.

ASG recommends that, whenever possible, you use MPXX to specify the literal delimiter, since this allows:

- Code optimization
- Improved error checking.

In corporate EXECUTIVE-ROUTINEs, the keyword DEFCMD may be used to specify that any PRIMARY COMMANDs that have been renamed using the command:

```
SET PRIMARY-COMMAND _ _ TO _ _ ;
```

are to recognized in the EXECUTIVE-ROUTINE by their original names.

Example

```
mpxx literal=#
```

NOP

NOP is a dummy directive that has no effect. It can be used, for example, in the INTERPRET directive.

PARSE

[illegible]

where:

expr is an expression

var-name is the name of a variable.

The PARSE ARG directive parses the input to an executive routine and assigns the words to variables. The PARSE VALUE directive parses the result of evaluating an expression and assigns the words to variables.

If there are more words than there are variables, the extra words are discarded. If there are fewer words than there are variables, the extra variables are set to null.

If the last name in the list of variables is of the form:

```
var-name ( )
```

then all remaining words are assigned to elements of this variable, starting from element 1. All other elements of the variable are set to null.

Delimiters are not removed when words are assigned. This is different from the way Parameter Variables are assigned. For example, if you invoke the executive routine MYEXEC as follows:

```
MYEXEC "i j";
```

the instructions:

```
PARSE ARG A()  
VLIST ARRAY A  
VLIST &P
```

produce the following output:

```
A (L)      00001      '"i j"'  
&P0        'i j'
```

Example 1

The instructions:

```
mpxx literal=#  
parseoption 4  
s = #11"22 3#  
parse value s with a()  
vlist array a
```

produce the following output:

```
A (L)      00001      '11"22'  
A (L)      00002      '3'
```

Example 2

Invoking the executive routine MYEXEC as follows:

```
MYEXEC abc d ef;
```

then the instructions:

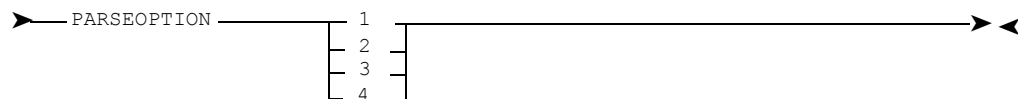
```
PARSE ARG FIRST(2) SECOND
VLIST LOCAL
```

produces the following output:

```
FIRST (L)    00002    'abc '
SECOND (L)   00001    'd '
```

The last word is discarded.

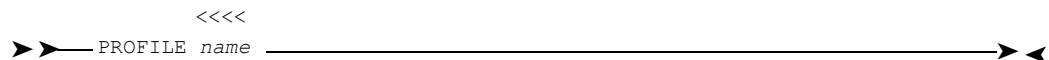
PARSEOPTION



The PARSEOPTION directive sets the parsing method. The new parsing method applies for the rest of the current executive routine, or until the PARSEOPTION directive is used again.

Refer to ["Parsing" on page 18](#) for further details of parsing.

PROFILE



where *name* is any valid user-defined variable name

The PROFILE directive declares profile variables.

When a PROFILE directive is encountered, each variable name is placed in the index of profile, global and command variables. Variables placed in the profile/global/command index by means of the PROFILE directive are accessible throughout the time a ControlManager user is logged on. If the PROFILE directive refers to a variable name which is already in the index, it has no effect on that variable other than to identify or re-identify it as a profile variable.

Profile variables are not lost at LOGOFF but are saved in a Variable Pool member on the MP-AID at LOGOFF provided that:

- The MP-AID is updateable
- The user is logged on under an exclusive logon profile or is the Systems Administrator.

They are then automatically retrieved from the MP-AID and restored at the next LOGON by this user.

RELEASE

The RELEASE command erases a selection of variables of a particular type. Use the RELEASE directive to erase:

- All variables of a particular type
- A selection of variables of a particular type

It is good programming practice to erase variables when you have finished with them.

The variable-type keywords have the following meanings:

Keyword	Meaning
COMMAND	User-defined command variables
GLOBAL	User-defined global variables
LOCAL	User-defined local variables
PROFILE	User-defined profile variables
&G	ASG-defined global variables
&I	Installation variables
&L	ASG-defined local variables
&P	Parameter variables

Local and parameter variables are local to an executive routine. That is, there may be other occurrences of any variable in higher-level executive routines. When a local or parameter variable is erased it is as if the most recent occurrence of that variable had never existed.

Variables of any other type are global. That is, any variable is the unique occurrence of that variable. When such a variable is erased it is as if that variable had never existed.

Erasing All Variables of a Particular Type

By including any of the variable-type keywords you can erase all variables of those types. For example, to erase all global variables (user-defined and ASG-defined), enter:

```
release &g global
```

Erasing a Selection of Variables of a Particular Type

By including any of the following variable-type keywords:

- COMMAND
- GLOBAL
- LOCAL
- PROFILE

you can erase:

- Those variables whose names fall within a given range. (Use the FROM ... TO keywords.)
- Those variables whose names start with a given string. (Use the ONLY keyword.)

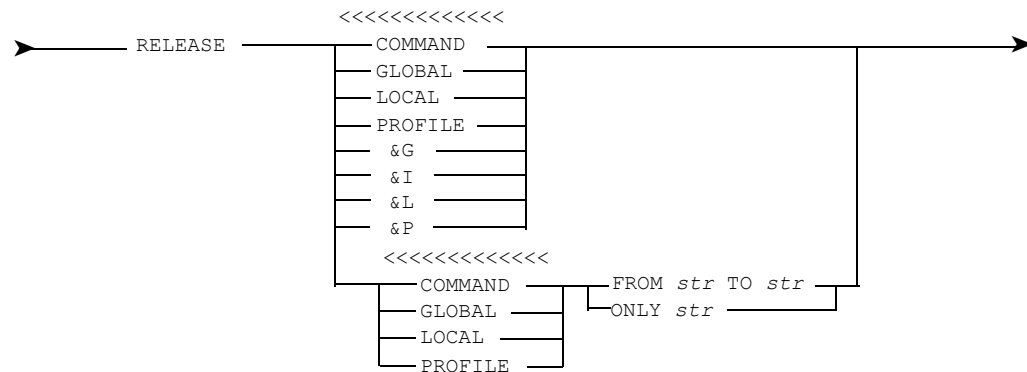
To erase all user-defined global or command variables whose names fall within the range a to d inclusive, enter:

```
release global command from a to d
```

To erase all user-defined global variables whose names begin with the string 'MDG_', enter:

```
release global only MDG_
```

RELEASE Syntax



where *str* is a string of up to 25 characters that is a user-defined variable name.

RETAIN

➤ ➤ — RETAIN ————— ➤ ➤

The RETAIN directive causes an executive routine to be retained in virtual storage when it terminates. In the absence of RETAIN the executive routine is deleted from virtual storage.

Note that RETAIN is only effective if the SET EXECUTIVE-RETENTION ON command has been issued.

RETURN

➤ ➤ — RETURN ————— ➤ ➤

The RETURN directive causes control to pass back to the most recent active CALL.

For full details of the RETURN directive, refer to the CALL directive.

SAY

```

>> SAY _____
      <<<<<<<<<
      expression

```

where *expression* is an expression subject to Full Evaluation.

The SAY directive is similar to the WRITEL directive, except that Full Evaluation takes place instead of Limited Evaluation.

The combined length of the evaluated expressions cannot exceed 598 characters. The maximum evaluated length of individual expressions cannot exceed 255 characters.

If you are using the Procedures Language facility in an interactive environment then setting BLANK-LINE-DISPLAY to ON causes the SAY directive to output a blank line if SAY is specified without any following text or if the expressions are evaluated to null strings or blanks.

SET

```

>> SET variable-name expression _____

```

where:

variable-name is the name of a variable

expression is an expression subject to Full Evaluation.

The SET directive assigns the result of an evaluated expression to the specified variable.

An alternate way of assigning values is to use the = operator. For example:

```
i = i+1
```

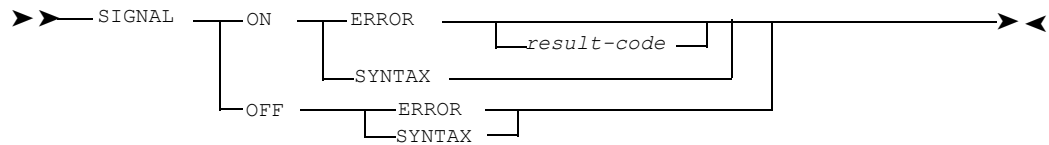
Examples

```

set i i+1
SET &G2 '17 JULY 1992'
SET &L20 LENGTH(PASSWORD)

```

SIGNAL



where *result-code* is an expression that evaluates to any integer between 0 and 255 inclusive. It defaults to 8.

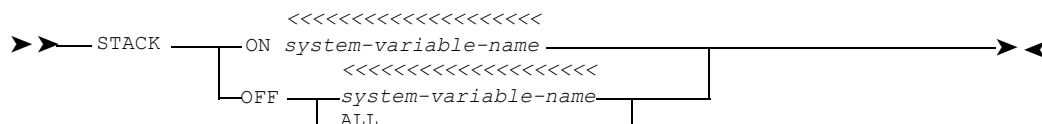
The SIGNAL directive allows you to divert control within an executive routine when an error occurs.

There are four cases.

- SIGNAL ON ERROR result-code
This turns on branching to the label -ERROR when a Manager Products command or executive routine returns a non-zero return code. Branching only occurs when this non-zero return code is greater than or equal to the result of evaluating result-code.
Manager Products commands return a return code of 0, 4 or 8. Executive routines return a return code between 0 and 255.
- SIGNAL OFF ERROR
This turns off branching to the label -ERROR.
- SIGNAL ON SYNTAX
This turns on branching to the label -SYNTAX when a directive fails.
- SIGNAL OFF SYNTAX
This turns off branching to the label -SYNTAX.

SIGNAL ON ERROR and SIGNAL ON SYNTAX can be active concurrently.

ASG recommends that the first instruction following the -ERROR label is SIGNAL OFF ERROR and the first instruction following the -SYNTAX label is SIGNAL OFF SYNTAX. This avoids the possibility of an infinite loop if an error occurs while processing the instructions following -SYNTAX or -ERROR.

&COLO, &CURL, &ENAM, &LINC, &LINO, **and** &PNUM

If stacking is switched ON for particular system variables, then all values subsequently assigned to those system variables are preserved until stacking is switched off or the current top level executive routine has terminated, whichever occurs first.

Normally, with stacking switched OFF, the previous value assigned to a system variable is replaced with a new value, whenever a new value is assigned by the Manager Products software. However, with stacking switched ON, an array is automatically generated by the Manager Products software, bearing the name of the system variable. On each occasion that a new value is assigned by the Manager Products software to the system variable, the new value is stored in the next unused element of the array so that all previous values are preserved.

The following example illustrates the effect of stacking a system variable. When a system variable, for example &MSNO, is first assigned a value, the value is assigned to element number one in array &MSNO, and may be referenced within an executive routine as &MSNO or &MSNO(1).

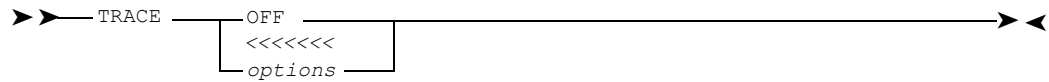
If stacking is OFF when the next value is assigned by the Manager Products software to system variable &MSNO, then that value overwrites the existing value in &MSNO(1).

However, if stacking is switched ON when the next value is assigned by the Manager Products software to system variable &MSNO, then that value is stored in the next unused element in array &MSNO which in this case is element 2 and so on. Element 2 in array &MSNO can be referenced as &MSNO(2).

When stacking is switched from ON to OFF for a particular system variable, any array values set up for that system variable are deleted except for the highest (that is, the most recent), which is reassigned to array element 1, and may be referenced within the executive routine as, for example, &MSNO or &MSNO(1).

143

TRACE



where *options* is one of these:

ALL	All lines are displayed as they are executed (equal to COMMAND)
ASSIGNS	Only assignment and DROP, PARSE, and RELEASE instructions are displayed
COMMAND	All lines are displayed as they are executed (equal to ALL)
COMMANDS	Only commands are displayed
CPUTIME	Displays CPU usage by command on a cumulative basis
LOGIC	Only IF, DO, END, ITERATE, LEAVE, CALL, RETURN, EXIT, and GOTO instructions, and labels are displayed
OFF	Tracing is turned off
ON	Is the same as the ALL option
RESULTS	Displays the results of all expression evaluations for instructions being displayed by other options

The TRACE directive allows you to display selected lines as an executive routine runs. Use it to debug executive routines.

TRACE RESULTS is equivalent to TRACE ALL RESULTS, when no other TRACE options are in force.

When an instruction is displayed it is prefixed by the line number. When a result is displayed (enclosed in single quotes) it is prefixed by >>>.

The TRACE directive is cumulative. For example:

```
TRACE COMMANDS
TRACE RESULTS
```

is equivalent to:

```
TRACE COMMANDS RESULTS
```

The effect of a TRACE instruction is limited to the current executive routine.

When output has been switched to an alternative or additional dataset (by means of the SWITCH OUTPUT command), TRACE output is sent only to the alternative dataset and not to the Primary Output Device.

A plus symbol (+) following the line number indicates that the displayed line is broken into two or more lines in the original executive routine.

For details on obtaining trace information without embedding TRACE directives in each routine, see the SET TRACE command.

Example

The instructions:

```
TRACE RESULTS
first = 'THIS IS A'
second = MESSAGE
SAY first second
```

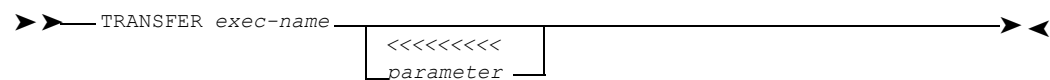
produce the following output:

```

LINE      2              first = 'THIS IS A'
>>> 'THIS IS A'
LINE      3              second = MESSAGE
>>> 'MESSAGE'
LINE      4              SAY first second
>>> 'THIS IS A'
>>> 'MESSAGE'
THIS IS A MESSAGE

```

TRANSFER



where:

exec-name is the name of a user or corporate executive routine

parameter is a parameter.

The TRANSFER directive:

- Terminates the current executive routine and all higher level executive routines
- Calls the named executive routine.

When the called executive routine terminates, control passes to command level. (Contrast this with a normal call of an executive routine, where when the called routine terminates, control passes back to the calling routine.)

The combined length of all the evaluated parameters cannot exceed 3999 characters. The parameters are made available to the executive routine in &P variables.

VLIST

The VLIST directive lists a selection of variables. It is especially useful when you are developing or debugging executive routines.

You can list:

- Particular variables
- All variables of a particular type
- Selected variables of a particular type.

The variable-type keywords have the following meanings:

Keyword	Meaning
COMMAND	User-defined command variables
GLOBAL	User-defined global variables
LOCAL	User-defined local variables
PROFILE	User-defined profile variables
&G	ASG-defined global variables
&I	Installation variables
&L	ASG-defined local variables
&SYS	System variables
&P	Parameter variables

When you list variables of a particular type:

- For ASG-defined variables (except system variables), all non-null variables are listed
- For user-defined variables, all variables are listed
- For system variables, all but the following variables are listed:

&BUFN, &COLO, &CURL, &ENAM, &LINC, &LINO and &PNUM.

Each line of the VLIST output has the following format:

- Variable name
- For user-defined variables, the variable type
- For arrays, the element number and the element's value enclosed in quotes
- For simple variables, the variable's value enclosed in quotes.

When output has been switched to an alternative or additional destination (by means of the SWITCH OUTPUT command), VLIST output is sent only to the alternative destination and not to the Primary Output Device.

Listing Particular Variables

To list particular variables use the ARRAY keyword. For example, to list all elements of the user-defined variable DB2_USERS, enter:

```
vlist array DB2_USERS
```

Listing All Variables of a Particular Type

By including any of the variable-type keywords you can list all variables of those types. For example, to list all global variables (user-defined or ASG-defined), enter:

```
vlist &g global
```

Listing a Selection of Variables of a Particular Type

By including any of the following variable-type keywords:

- COMMAND
- GLOBAL
- LOCAL
- PROFILE

you can list:

- Those variables whose names fall within a given range. (Use the FROM ... TO keywords.)
- Those variables whose names start with a given string. (Use the ONLY keyword.)

To list all user-defined global or command variables whose names fall within the range 'a' to 'd' inclusive, enter:

```
vlist global command from a to d
```

To list all user-defined global variables whose names begin with the string MDG_, enter:

```
vlist global only MDG_
```

Examples

The instructions:

```
mpxx literal=:  
&L12 = :YELLOW:  
BOX(3) = :CLOSED:  
BOX(6) = :OPEN:  
GLOBAL BOOK CASE  
BOOK = :EMPTY:
```

declare and initialize some variables, and the instruction:

```
VLIST &L GLOBAL ARRAY BOX
```

produces the following output:

```
&L12      'YELLOW'  
BOOK (G)   00001   'EMPTY'  
CASE (G)                   INDEX ENTRY ONLY  
BOX (L)    00003   'CLOSED'  
BOX (L)    00006   'OPEN'
```

The instructions:

```
STACK ON &MSNO  
MPR SET OUTPUT-EDIT OFF ;  
EDIT ENTITY ;  
VLIST ARRAY &MSNO
```

produce the following output:

```
DM08100I          SET PROCESSING SUCCESSFUL  
DM08816E          MP-AID USER MEMBER ENTITY NOT PRESENT  
&MSNO (S)   00001   '8100'  
&MSNO (S)   00002   '8816'
```


You can use the WRITEF directive in a similar way to the SAY directive. However, whereas SAY outputs text to the primary/secondary output device only, WRITEF can output text to a destination specified by a previous SENDF command. This destination can be:

- A USER-MEMBER on the MP-AID
- A sequential dataset
- A partitioned dataset

and/or the primary/secondary output device. If no destination has been previously specified, WRITEF will output to the primary/secondary output device only.

The maximum length of data of an expression after evaluation is 32760 characters, depending on the output destination.

The maximum length of data that can be output to a USER-MEMBER on the MP-AID is 255 bytes.

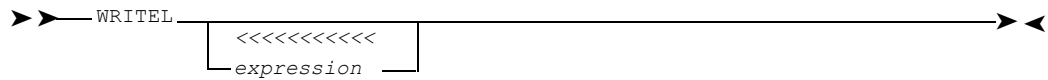
The maximum length of data that can be written to an external dataset is the smallest value of:

- The logical record length of the dataset, for fixed length records; or
- The logical record length of the dataset, minus four bytes, for variable length records; or
- 32760

If the WRITEF directive is followed by expressions whose combined evaluated length exceeds the maximum size for the destination being written to, the executive routine terminates with an error message (unless the SIGNAL ON SYNTAX directive is active).

A WRITEF directive which follows a CLOSEF directive generates output to the destination specified by the last active SENDF command.

WRITEL



where *expression* is an expression subject to Limited Evaluation.

The WRITEL directive outputs text to the Primary Output Device.

The WRITEL directive is similar to the SAY directive except that Limited Evaluation takes place instead of Full Evaluation.

There is a restriction of 598 characters both on the number of text characters (including imbedded blanks) which can be specified with a single WRITEL directive and on the output text generated by a single WRITEL directive.

If you are using the Procedures Language facility in an interactive environment then setting BLANK-LINE-DISPLAY to ON will cause the WRITEL directive to output a blank line if WRITEL is specified without any following text or if the text is evaluated to a null string or blanks.

Example 1

```
WRITEL DICTIONARY &DICT NOW OPEN
```

outputs the following text, if &DICT is set to DEMO:

```
DICTIONARY DEMO NOW OPEN
```

Example 2

```
WRITEL &L1+&L4 ||12345 ||ABCDE
```

produces the following output, if &L1 is set to 5 and &L4 is set to 7:

```
5+712345ABCDE
```

9

Functions

This chapter contains specifications, in alphabetical order, of all functions.

ABBREV	155
ARG	156
ARRAYHI	157
ARRAYLO	157
BIN	158
CENTER	158
CLIENTI	158
CLIENTN	159
CLIENTU	159
COPIES	160
DB2TYPE	160
DIVCAPT	160
DIVOBJN	161
DIVOBJT	161
EDDATE	161
EDTIME	163
EXTRACT	163
FDO	166
GETSVRM	167
GETTOKEN	168
GETUDSN	168
HEX	169
INSERT	169
LASTPOS	170
LEFT	170

LENGTH	170
LOWER	171
MAX	172
MEMTYPE	172
MIN	173
MPRAID	173
MPRCMPW	174
MPRDDPW	174
MPRSU	174
MPRUCLS	174
MPRUDSN	175
NDATE	175
NTIME	176
OVERLAY	177
PACK	177
PARSABLE	178
POS	178
PTIME	178
REDUCE	179
REPSTR	179
REVERSE	180
RIGHT	180
ROOT	180
SEARCH	181
SERVERN	182
STIME	182
STRIP	182
SUBSTRING	184
SUBTASK	185
SUBTENV	186
TRANSLAT	187
TRUNCATE	187
TYPE	188

UPPER.....	189
VALUE.....	190
WORD	191
WORDINDX	191
WORDLEN	191
WORDS	192

ABBREV

➤ ➤ ABBREV(*string* , *strg* , *length*) ➤ ➤

The ABBREV function tests whether *strg* is a substring of a second string *string*.

- ABBREV returns 1 if *strg* is equal to the leading characters of *string* and *strg* is not shorter than a minimum specified *length*.
- ABBREV returns 0 if either condition is not true.
- *strg* defaults to null.
- *length* defaults to the length of *strg*.

Examples

```

ABBREV('Heading','Head')   gives  1
ABBREV('HEADING','Head')   gives  0
ABBREV('HEADING','HEA',4)  gives  0
ABBREV('HEADING','')       gives  1
ABBREV('HEADING','','1)    gives  0

```

The function is particularly useful in identifying truncated forms of keywords supplied by the user as input parameters to an executive routine, for example:

```

LITERAL #
IF &P0 EQ '' THEN GOTO ERROR1
IF ABBREV(SELECT,&P0,1) EQ 1 THEN GOTO SELECT
IF ABBREV(#MOD-LEVEL#,&P0,1) EQ 1 THEN GOTO MODIFY
IF ABBREV(VERIFY,&P0,3) EQ 1 THEN GOTO VERIFY

```

ARG

➤➤ ARG (variable-number ,variable-type) ➤➤

The ARG function returns the value of the specified ASG-defined variable.

For example, if *variable-number* is 5 and *variable-type* is G then the value of global variable &G5 is returned.

If *variable-number* is omitted the value returned by the function is the highest variable number that has a non-null value assigned to it, plus 1.

variable-number, if specified, may be any number from 0 up to 99 (the maximum number permitted for ASG-defined variables).

variable-type is one of the following:

G representing &G variables
 I representing &I variables
 L representing &L variables
 P representing &P variables

If *variable-type* is omitted P is assumed.

Examples

In the following example, &P0 to &P4 have been set up, including &P2 set to ABCDE:

```
SAY ARG(2)           gives  ABCDE
SAY ARG(,P)          gives  5
```

The ARG function makes it possible to loop through a sequence of ASG-defined variables without referring to each by name, for example:

```
set N 0                /* number of variable
set count arg()         /* get number of &p variables
if count eq 0 then exit /* exit if no &p vars
-loop
if arg(N,) eq avalue then do
  .... other instructions ....
end
set N N+1              /* increment for next variable
if N lt count then goto loop /* exit if there are no more
exit
```

ARRAYHI

➤ ➤ — ARRAYHI (*expression*) ————— ➤ ➤

The ARRAYHI function evaluates the specified expression, which must be a user-defined variable name. If the variable name is found and data has already been assigned to it, the highest array number set up is returned.

If the variable name is not found in any user variable index, the value returned by ARRAYHI is -1.

If the variable name is found but no data has been assigned to it, the value returned is zero.

Example

```
LITERAL #
FRED(6) = BLOB
FRED(10) = LUMP
SAY ARRAYHI (#FRED#)
```

gives a value of 10.

ARRAYLO

➤ ➤ — ARRAYLO (*expression*) ————— ➤ ➤

The ARRAYLO function evaluates the specified expression, which must be a valid user-defined variable name. If the variable name is found and data has already been assigned to it, the lowest array number set up is returned.

If the variable name is not found in any user variable index, the value returned by ARRAYHI is -1.

If the variable name is found but no data has been assigned to it, the value returned is zero.

Example

```
LITERAL #
FRED(6) = BLOB
FRED(10) = LUMP
SAY ARRAYLO (#FRED#)
```

gives a value of 6.

BIN

►—BIN(*integer*) —————►◀

where *integer* is a value between -999999 and +999999.

The BIN function returns a word containing the signed binary representation of the numeric argument. The purpose of this function is to enable binary fields to be included within WRITEF directive output. No computational capability is provided.

Examples

```
BIN(1000)      gives '000003E8'
BIN(32767)     gives '00007FFF'
BIN(-1)        gives 'FFFFFFFF'
```

CENTER

►►—CENTER (*string,length*) —————►◀
 CENTRE (,*pad*)

The CENTER function returns a string that is centered in a field of length *length*. *pad* characters are added as necessary to make up the length. The default pad character is blank. If *string* is longer than *length* it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right end loses or gains one more character than the left end.

Examples

```
CENTER(MID,7)           gives '  MID  '
CENTER(MID,8,'*')       gives '**MID**'
CENTER('1st cent pos',8) gives 't cent p'
CENTRE('2nd cent pos',7) gives 'd cent '
```

CLIENTI

►►—CLIENTI () —————►◀

The CLIENTI function returns the current MPSF client identity. This is an alphanumeric string of up to 8 characters.

Example

```
clienti()
```

CLIENTN

➤ ➤ — CLIENTN() ————— ➤ ➤

The CLIENTN function returns the current MPSF client conversation number in the range of 1 to 9,999.

-1 is returned if the function is issued from a non-MPSF environment.

-3 is returned if any argument is passed.

Example

```
clientn()
```

CLIENTU

➤ ➤ — CLIENTU() ————— ➤ ➤

For MPSF OS/390 clients only, the CLIENTU function returns the client RACF/ACF2 user ID under which the client job or online session is being executed.

-1 is returned if the caller is not executing as an MPSF OS/390 client.

-3 is returned if any argument is passed.

Example

```
clientu()
```

COPIES

➤ ➤ — COPIES (*string*, *number*) ————— ➤ ➤

The COPIES function returns number concatenated copies of string. number must be equal to or greater than 0.

Examples

```
COPIES (MANY, 3)    gives  'MANYMANYMANY'
COPIES (MANY, 0)    gives  ''
```

DB2TYPE

The DB2TYPE function is used internally by Manager Products. It is intended in future to make it available as part of the procedures language. At present, however, DB2TYPE is a reserved name.

If you want to use a variable DB2TYPE then in order for it to be correctly interpreted as a variable it must be enclosed in delimiters.

DIVCAPT

➤ ➤ — DIVCAPT (*resource*, *request*) ————— ➤ ➤

where *request* is one of these:

- a to return the current attempted CAPTURE count
- r to return the current actual CAPTURE count
- i to return the current CAPTURE interval in seconds

The DIVCAPT function returns information about a CAPTURE session for a DIV repository or MPAID resource defined and opened in shared mode under MPSF.

-1 is returned if a CAPTURE session is not active for the specified resource.

-2 is returned if the specified resource is not found.

-3 is returned if the function is issued from a non-MPSF environment.

-4 is returned if invalid or incorrect arguments are passed.

Examples

```
divcapt (prod, a)
divcapt (admin, r)
divcapt (mpaid, i)
```

DIVOBJN

➤ ➤ — DIVOBJN () ————— ➤ ➤

When issued by an MPSF RPT subtask, the DIVOBJN function returns the name of the owned DIV object.

-1 is returned if no DIV object is owned.

Example

```
divobjn ( )
```

DIVOBJT

➤ ➤ — DIVOBJT () ————— ➤ ➤

When issued by an MPSF RPT subtask, the DIVOBJT function returns the type of the owned DIV object.

M is returned if the DIV object is an MPAID.

R is returned if the DIV object is a repository.

-1 is returned if no DIV object is owned.

Example

```
divobjt ( )
```

EDDATE

➤ ➤ — EDDATEx (*expression*) ————— ➤ ➤

where x is I or O representing input and output format respectively.

The EDDATE functions are used to reformat the ASG standard date format into your installations standard format, as defined in the DCUST installation macro.

The standard ASG date format is YYYYDDD, where:

- Y represents a year character
- D represents a day character.

For example, the 3rd February 1989 would be represented as 1989034.

The EDDATEI function gives the date for input to a Manager Products repository or command.

The EDDATEO function gives the date as output by a Manager Products command.

Example

```
literal #  
eddatei (#1991365#)      gives      31/DEC/1991
```

if you are using the default date format settings as supplied, and user definable via operands of the DCUST macro.

EDTIME

➤ ➤ — `EDTIMEn(expression)` ————— ➤ ➤

where *n* is I or O representing input and output format respectively.

The EDTIME functions are used to reformat the ASG standard time format into your installations standard format, as defined in the DCUST installation macro.

The standard ASG time format is HHMMSSSTH where:

- H represents an hour character
- M represents a minute character
- S represents a second character
- TH represents two thousands of a second characters.

For example, 4 hours, 26 minutes, 37 seconds and 352 thousands of a second would be represented as 04263735.

The EDTIMEI function gives the time for input to a Manager Products repository.

The EDTIMEO function gives the time for output from a Manager Products repository

Example

```
literal #
edtimei(#11200501#)    gives    11.20.05
```

if you are using the default time format settings as supplied, and user definable via operands of the DCUST macro.

EXTRACT

➤ ➤ — `EXTRACT(keyword)` ————— ➤ ➤

where *keyword* is one of the keywords listed below.

The EXTRACT function returns information about your Manager Products environment.

The valid keywords, and the information returned by EXTRACT in each case, are shown in the table below.

Keyword	Information Returned
CMD	Most recent command-line data as entered
CMRREL	Current ControlManager version/release number
CPUTIME	Cumulative CPU time used, in seconds
DFREE	Current Data Entries dataset free block count
DMRREL	Current DataManager version/release number
DSN	Physical dataset name (if allocated)
DSRREL	Current DesignManager version/release number
DYRREL	Current DictionaryManager version/release number
ELEVEL	Level of the current executive routine
ENAME	Name of the current executive routine
ETYPE	Type of current executive routine (M,C,U or T)
EYRREL	Current DictionaryManager version/release number
IFREE	Current Index dataset free block count
LCOFF	Offset down the ControlManager buffer where linear command was entered
LFREE	Current free log space in kilobytes
MFREE	Current MP-AID dataset free block count
MMRREL	Current MethodManager version/release number
MPUPDC	Current primary MPAID update count
NOPRINT	1 when NOPRINT exec-name is specified, otherwise 0
PAGELN	Current MPR page length
PROFID	Current profile identifier
REPUPDC	Current repository update count, -1 if not open
SFREE	Current Source dataset free block count
SRBTIME	Cumulative Service Request Block time, in seconds
VERB	Most recent command verb entered, unabbreviated

The CPUTIME keyword is only available in MVS and VM environments. The function call:

```
EXTRACT (CPUTIME, C)
```

returns the CPU time used since the start of the highest level executive routine.

The SRBTIME keyword is only available in MVS environments.

DSN Keyword

The DSN keyword allows writers of executive routines to determine if a particular ddname has been allocated and, if it has, the physical dataset name allocated. The specification is:

```
EXTRACT (DSN, ddname)
```

where *ddname* is up to 8 characters.

If the dataset has not been allocated a null string is returned. If the dataset has been allocated the following is returned:

- Under MVS, a name of up to 44 characters
- Under VM/CMS, an 18 character string consisting of the file name-type-mode

For a concatenated dataset, only the first dataset name in the concatenation is returned.

Primary Command Keyword

This keyword allows the EXTRACT function to return details about a passed primary command verb, reporting availability and current name.

The syntax of this option is as follows:

```
EXTRACT (PRIMARY, command)
```

where *command* is a Manager Products command verb as supplied by ASG.

Returned values are:

- | | |
|----------|--|
| 0 string | Command available / current command string |
| 1 string | Command disabled / current command string |
| 2 | Passed command ambiguous |
| 3 | Passed command not found in command table |

Examples

Assuming the Manager Products COPY command has been renamed as DUPLICATE then:

```
EXTRACT (PRIMARY, COPY)
```

returns

```
0DUPLICATE.
```

Assuming the Manager Products UPDATE command has been disabled then:

```
EXTRACT (PRIMARY, UPDATE)
```

returns

```
1UPDATE.
```

LCOFF Keyword

This keyword allows a currently executing linear command to determine the offset down the ControlManager buffer where the linear command was entered. The buffer is considered to consist of the user data plus the top and end of data records.

Where not issued from a linear command, -1 is returned.

FDO

➤ ➤ — FDO (*keyword*) ————— ➤ ➤

where *keyword* is one of the keywords listed below.

The FDO function returns information about currently active blocks.

These are the valid keywords and the information returned by FDO in each case:

Keyword	Information Returned
DARRAY	The current array element number, if the currently active block is a DO array-name() block, otherwise zero.
DFOR	The number of times the currently active block has been executed. (1 during the first execution, 2 during the second execution, and so on.)
DLEVEL	The number of currently active blocks.

If there are no active blocks FDO returns zero.

Example

The instructions:

```
if 1 = 1 then do
  say fdo(dlevel)
  if 1 = 1 then do
    say fdo(dlevel)
  end
end
end
```

produce the following output:

```
1
2
```

GETSVRM

➤ ➤ — GETSVRM (*msg-no*) ————— ➤ ➤

where *msg-no* must be an integer in the range 1 to 50.

The GETSVRM function returns the text of message strings which can be defined by the System Administrator for access by all MPSF clients. Each returned string can be up to 255 characters in length.

A null string is returned when no message exists.

-1 is returned if the function is issued from a non-MPSF environment.

-3 is returned if an invalid argument is passed.

Examples

```
getsvrm(1)
getsvrm(50)
```

GETTOKEN

➤ ➤ — GETTOKEN () ————— ➤ ➤

When issued by an MPSF client, the GETTOKEN function returns an integer in the range 1 to 2,147,483,647. The returned integer is always unique across all clients for the duration of MPSF. You can use this function to generate a unique resource name, such as MPAID user member name.

-1 is returned if the function is issued from a non-MPSF environment.

Example

```
gettoken()
```

GETUDSN

➤ ➤ — GETUDSN(repository) ————— ➤ ➤

The GETUDSN function returns the UDS table name of a DIV repository defined and opened as a shared repository under MPSF.

-1 is returned if the specified repository is not defined and opened as a shared repository.

-2 is returned if invalid or incorrect arguments are passed.

-3 is returned if the function is issued from a non-MPSF environment.

Example

```
getudsn(admin)
```

HEX

➤ ➤ — `HEX(string)` ————— ➤ ➤

The HEX function returns a string containing the hexadecimal representation of the passed argument. The maximum length of the argument is 127 characters.

-1 is returned if the passed argument is a null string

-2 is returned if invalid or incorrect arguments are passed.

-3 is returned if the passed argument contains more than 127 characters.

Examples

```
hex(ABCD)  gives C1C2C3C4
hex(12345) gives F1F2F3F4F5
```

INSERT

➤ ➤ — `INSERT(new,target options)` ————— ➤ ➤

where *options* is:

➤ ————— ➤ ➤

┌ , ───┐ ┌───┐ ┌───┐ ┌───┐ ┌───┐

└───┐ └───┐ └───┐ └───┐ └───┐

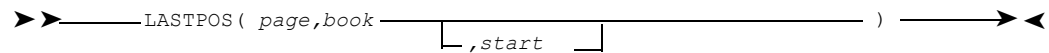
n *length* *,pad*

The INSERT function inserts the string *new* into the string *target* after the *nth* character. The string *new* is padded to *length* with the character specified by *pad*, or truncated if *length* is less than the length of *new*. If *n* is set to the default of zero then the string *new* is inserted before the string *target*. *length* defaults to the length of *new*. The default *pad* character is a blank.

Examples

```
INSERT('','newgap',3)           gives 'new gap'
INSERT('tst','gap',5,6)         gives 'gap tst '
INSERT('tst','gap',5,6,'+')     gives 'gap++tst+++'
INSERT('tst','gap')             gives 'tstgap'
INSERT('tst','gap',,5,'-')      gives 'tst--gap'
```

LASTPOS



The LASTPOS function returns the position of the last occurrence of one string, *page*, in another string, *book*. Searching begins at character position *start* and scans from right to left. By default the search starts at the last character of *book*. 0 is returned if *page* is not found in *book*.

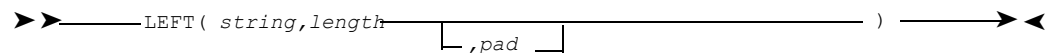
Examples

```

LASTPOS('','tes tst ring')    gives    8
LASTPOS('','teststring')     gives    0
LASTPOS('','tes tst ring',7)  gives    4

```

LEFT



The LEFT function returns a string of length *length* containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters on the right as needed. If *length* is zero the null string is returned.

pad defaults to blank. *length* must be zero or a positive integer.

Examples

```

LEFT('links',8,'.')    gives    'links...'
LEFT('lagauche',7)     gives    'lagauch'

```

LENGTH

There are two versions of the length function:

- LENGTH function
- Implicit length function.

ASG recommends that you use the LENGTH function. The implicit version is retained for upwards compatibility only.

The two versions are described below.

LENGTH function

►►——LENGTH(*string*) ——►◀

The LENGTH function returns the length of string *string*.

Example

```
literal #
reply = #ABCD#
SAY LENGTH(reply)           gives    4
SAY LENGTH(reply)+LENGTH(reply) gives    8
```

Implicit Length Function

►►——LENGTH *string* ——►◀

The LENGTH function returns the length of string *string*.

Example

```
literal #
reply = #ABCD#
SAY LENGTH reply           gives    4
```

LOWER

►►——LOWER(*string*) ——►◀

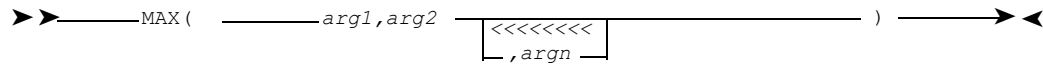
where *string* is a string.

The LOWER function returns the string *string* with all characters translated to lower case.

Examples

```
literal #
lower(#A#)      gives    a
lower(#abCDe#)  gives    abcde
lower(#123#)    gives    123
```

MAX



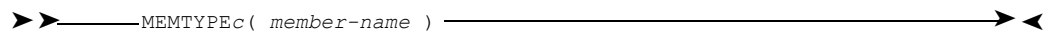
The MAX function returns the largest value from the list of passed arguments.

A minimum of two arguments must be passed, but there is no maximum number of arguments.

Example

MAX(100, 300, 500, 1000) gives 1000

MEMTYPE



The MEMTYPE functions take a single parameter, a member name, and return that member's member-type in the form requested.

You specify the required form of the member-type by varying the last letter of the function name, as shown below.

Last Letter	Information Returned
B	The base member-type keyword
E	The first encode keyword
I	The first interrogate keyword
L	The standard literal
P	The plural literal
R	The first report-down-to keyword
S	The short literal
X	The long literal

These member-type literals and keywords are defined in the definition of the member-type in the UDS table. If any of the member-type literals are not defined, the functions return the standard literal, which is mandatory.

Example

```
literal #
memtypes(#PROCL2500#)    gives    INFO PANEL
```

if the current UDS table is DU001 and PROCL2500 is an INFOBANK-PANEL.

MIN

Diagram illustrating the function signature for `MIN`. The function name `MIN` is followed by a list of arguments: `arg1, arg2` and a variable number of arguments indicated by `...` and `argn`. The arguments are enclosed in parentheses. The diagram uses arrows to indicate the flow of data from the arguments to the function name.

The MIN function returns the smallest value from the list of passed arguments.

A minimum of two arguments must be passed, but there is no limit to the maximum number of arguments which may be passed.

Example

MIN(100,300,500,1000) returns 100

MPRAID

➤➤————MPRAID ()————➡➤

The MPRAID function returns a numeric value, which indicates whether the current top level executive routine was initiated by input from the Command Area and/or a PF key. The significance of the numeric values which may be returned are given in the table below:

MPRAID Value	Command Line Input	PFkey Input
1 to 24	No	Yes
100	Yes	No
101 to 124	Yes	Yes

The MPRAID value identifies the particular PF key pressed, if applicable.

If the MPRAID value is 100 then data entry was initiated by means of the ENTER key.

If MPRAID is any value other than 100 then data entry was initiated by a PF key.

MPRCMPW

➤ ➤ _____MPRCMPW () _____➤ ➤

The MPRCMPW function returns the password for the current Manager Products Logon session.

MPRDDPW

➤ ➤ _____MPRDDPW () _____➤ ➤

The MPRDDPW function returns the password for the current repository, if any.

MPRSU

➤ ➤ _____MPRSU (*su-code*) _____➤ ➤

where *su-code* is a four character Selectable Unit Code.

The MPRSU function returns 1 if the specified Selectable Unit code is present at your installation, or 0 if the selectable unit code is absent at your installation.

MPRUCLS

➤ ➤ _____MPRUCLS (*current-user-class*) _____➤ ➤

where *current-user-class* evaluates to one of these letters:

- A or a identifying systems administrator
- C or c identifying repository Controller
- G or g identifying guest Controller
- M or m identifying Master Operator
- U or u identifying general user

The MPRUCLS function returns the value 1 if the current user belongs to the specified current-user-class, otherwise 0 is returned.

A user may belong to more than one current-user-class at the same time. For example, a user may be a systems administrator and a guest Controller.

MPRUDSN

➤ ➤ _____MPRUDSN () _____ ➤ ➤

The MPRUDSN function returns the UDS Table name of the current repository.

NDATE

➤ ➤ _____NDATEx(*expression*) _____ ➤ ➤

where *x* is I or O representing input and output format respectively.

The NDATE functions are used to reformat the date from your installations standard format, as defined in the DCUST installation macro, to the ASG standard date format.

The standard ASG date format is YYYYDDD, where Y represents a year character and D represents a day character, for example the 3rd February 1989 would be represented as 1989034.

The NDATEI function converts a date for input by a Manager Products command to the ASG standard date format.

The NDATEO function converts a date as output by a Manager Products command to the ASG standard date format.

Example

```
literal #
ndatei(#31 DEC 1991#)      gives      1991365
```

if you are using the default date format settings as supplied, and user definable via operands of the DCUST macro. (31 DEC was the 365th day of 1991.)

NTIME

➤ ➤ `NTIME n (expression)` ➤ ➤

where n is I or O representing input and output format respectively.

The NTIME functions are used to reformat the time from your installations standard format, as defined in the DCUST installation macro, to the ASG standard time format.

The standard ASG time format is HHMMSS TH where:

- H represents an hour character
- M represents a minute character
- S represents a second character
- TH represents two thousands of a second characters.

For example, 4 hours, 26 minutes, 37 seconds and 352 thousands of a second would be represented as 04263735.

The NTIMEI function converts the time for input to a Manager Products repository to the ASG standard time format.

The NTIMEO function converts the time for output from a Manager Products repository to the ASG standard time format.

Example

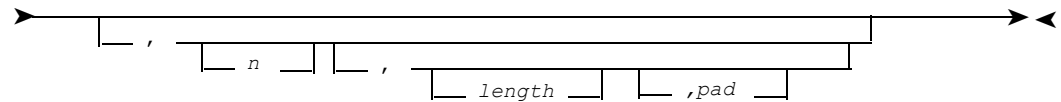
```
literal #  
ntimeo(#11.20.05.01#)      gives      11200501
```

if you are using the default time format settings as supplied, and user definable via operands of the DCUST macro.

OVERLAY

➤➤——OVERLAY(*new,target,options*) —————➤➤

where *options* is:



The OVERLAY function returns the string *target* overlaid by the string *new*, where *new* is padded or truncated to *length*, starting at character position *n* within *target*. If *n* is beyond the end of *target*, *target* is first padded out to position *n*.

n defaults to 1. If set, it must be a non-negative number. *length* defaults to the length of *new.pad* defaults to blank.

Examples

OVERLAY ('', 'bneath', 3)	gives	'bn ath'
OVERLAY ('', 'bneath', 3, 2)	gives	'bn. th'
OVERLAY ('ov', 'unda')	gives	'ovda'
OVERLAY ('ov', 'unda', 4)	gives	'undov'
OVERLAY ('ov', 'unda', 5, 6, '+')	gives	'und+ovr+++'

PACK

➤ ——— `PACK(integer)` ————— ➤ ➤

where *integer* is a value between -999999 and +999999.

The PACK function returns a word containing the packed decimal representation of the numeric argument. The purpose of this function is to enable packed decimal fields to be included within WRITEF directive output. No computational capability is provided.

Examples

```
PACK(99)           gives '0000099C'
PACK(1234567)      gives '1234567C'
PACK(-50)           gives '0000050D'
```

PARSABLE

➤➤——PARSABLE(*str*)——→➤➤

where *str* evaluates to a string.

The PARSABLE function tells you if the string can be parsed under the current parsing method. It returns 1 if the string can be parsed and 0 otherwise.

Refer to ["Parsing" on page 18](#) for further details of parsing.

POS

➤➤——POS(*page*,*book* , *start*)——→➤➤

The POS function returns the position of the first occurrence of the string *page* in another string *book*. Searching begins at character position *start* and scans from left to right. By default the search starts at the first character of *book*. 0 is returned if *page* is not found in *book*.

Examples

POS('cat','location')	gives	3
POS('cat','notfound')	gives	0
POS(' ','not mis sing')	gives	4
POS(' ','not mis sing',5)	gives	8

PTIME

➤➤——PTIME()——→➤➤

The PTIME function returns the time in *hh.mm.ss.ttt* format (*ttt* is thousandths of a second).

REDUCE

```
➤ ➡ REDUCE( name,reduction-length _____ ) ➡ ➤
```

└──,separator ─┘

The REDUCE function returns a reduced version of *name*. The first three characters of the returned string are a return code. The normal return code is three zeroes.

name is a string of 1 to 80 characters. *reduction-length* is an integer between 1 and 50. If *reduction-length* is greater than the length of *name* then the original name is returned unchanged. *separator* is a character that appears in name as a separator. It has no default value.

REDUCE is typically used to reduce a member name so that it is valid in an external environment. See the publication *ASG-Manager Products Relational Technology Support: SQL/DS* for the rules of the reduction process.

Examples

```
literal #
reduce(#FREDERICK#,4) gives 000FRED
reduce(#SPECIAL-ORDER-DATE-MONTH#,15,#-#) gives 000SP-ORD-DAT-MONT
```

REPSTR

►► — REPSTR(*str1 str2* [,*str3*]) **►◄**

The REPSTR function replaces all occurrences of *str2* in *str1* by *str3*. Any of the arguments can be null. *str3* defaults to null.

Examples

REPSTR (aabbcc,bb,xx)	gives	aaxxcc
REPSTR (aabbcc,bb)	gives	aacc
REPSTR (12341234,4,99)	gives	1239912399
REPSTR ('',' ',xyz)	gives	xyz

REVERSE

➤➤—`REVERSE(string)` —————➤➤

The REVERSE function returns *string*, reversed end-to-end.

Example

`REVERSE('back-to-front')` gives `'tnorf-ot-kcab'`

RIGHT

➤➤—`RIGHT(string,length —————)` —————➤➤
└──,*pad* ─┘

The RIGHT function returns a string of *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters on the left as needed. If *length* is zero the null string is returned.

pad defaults to blank. *length* must be zero or a positive integer.

Examples

`RIGHT('rechts',8,'.')` gives `'..rechts'`
`RIGHT('ladroite',7)` gives `'adroite'`

ROOT

➤➤—`ROOT(number,nth —————)` —————➤➤
└──,*round* ─┘

The ROOT function returns the *nth* root of *number*, rounded up or down if necessary to the nearest integer. (Up and down refer to the absolute value of the integer; that is, its sign is not taken into account).

number is a value between -1073741824 and +1073741824. *nth* is a value between 1 and 15. *round* is d or D (for round result down) or u or U (for round result up). The default value for *round* is D. When *number* is negative, *nth* must be odd. That is, calculation of imaginary roots is not available.

Examples

```

ROOT(1728,3,d)      gives  12
ROOT(-1730,3,u)     gives -13

```

SEARCH

➤➤ `SEARCH(variable, string options)` ➤➤

where *options* is:

➤ `[, [low] [, [high] [, c]]` ➤

The SEARCH function searches the array *variable* for an array element value matching *string*. It returns the number of the first such array element it finds, or 0 if the search fails. Searching begins at array element *low* and ends at array element *high*.

If *c* is set to M or m then string must equal the array element value for a match to be found. If *c* is set to any other value then string need only be contained in the array element value for a match to be found.

low defaults to the lowest array element number assigned. *high* defaults to the highest array element number assigned. *c* defaults to null.

If string evaluates to null then SEARCH returns the number of the lowest unassigned array element in the specified range.

Examples

```

LITERAL #
FRED(3) = #first string#
FRED(4) = #first#
FRED(6) = #second#
SEARCH(#FRED#,'first',,,m)      gives  4
SEARCH(#FRED#,'first')          gives  3
SEARCH(#FRED#,'first',4)        gives  4
SEARCH(#FRED#,'thirst')         gives  0
SEARCH(#FRED#,'',3)             gives  5

```

SERVERN

➤ ➤ — SERVERN () ————— ➤ ➤

The SERVERN function returns the name of the current MPSF server.

-1 is returned if the function is issued from a non-MPSF environment

-2 is returned if any argument is passed.

Example

```
servern()
```

STIME

➤ ➤ — STIME () ————— ➤ ➤

The STIME function returns the current time in hundredths of a second measured from midnight and provides a convenient way to calculate elapsed times.

-1 is returned if any argument is passed.

Example

```
stime()
```

STRIP

➤ ➤ — STRIP(*string options*) ————— ➤ ➤

where:

string is a string

options is:

➤ ————— ➤

— , —	— 'L' —	— , character —
	— 'T' —	
	— 'B' —	

character is a single character.

The STRIP function removes leading, trailing, or both leading and trailing characters from string *string*. The letters L, T and B have the following meanings:

- L To strip out leading characters
- T To strip out trailing characters
- B To strip out both leading and trailing characters.

If you omit the argument the default is B.

character specifies the character to be removed, with the default being a blank.

Examples

```
STRIP(' wash ')      gives 'wash'
STRIP(' poker ','L')  gives 'poker '
STRIP(' search ','T') gives ' search'
```

SUBSTRING

There are two versions of the substring function:

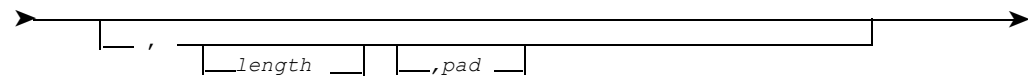
- SUBSTR function
- Implicit substring function.

ASG recommends that you use the SUBSTR function. The implicit function is retained for upwards compatibility only.

SUBSTR Function

➤➤ SUBSTR(*string*, *start options*) ➤➤

where *options* is:



The SUBSTR function returns the string of length *length*, padded as necessary by *pad* characters, starting at position *start* of string *string*. *length* defaults to the number of characters from position *start* to the end of *string*. *pad* defaults to blank.

Examples

SUBSTR('BOXES', 2, 2)	gives	'OX'
SUBSTR('BOXES', 3, 2)	gives	'XE'
SUBSTR('BOXES', 2, 6, ' .')	gives	'OXES..'

Implicit Substring Function

An implicit substring call is of the form:

variable-name(*start*, *length*)

where *variable-name* is the name of a variable, for example:

size(2,3)

or

USERVAR(A)(B,C)

where a substring is obtained of the value of element A in array USERVAR.

Examples

DAVE (2,2) gives AV

where DAVE is a literal.

```
&L1 = &L2 (&L4, &L3) + &G20 + 100
&L1 = &L3 (5, &L5 (3, 1))
```

SUBTASK

This is applicable only when a subtasking environment has been established. This function returns details of currently attached Manager Products tasks.

SUBTASK Function

➤➤——SUBTASK(*argument1*, *argument2*) —————➤➤

where valid values for *argument1* are:

CCOD	Returns the current value of the &CCOD system variable
COMMAND	Returns the value of the current command sent to the task
CPUTIME	Returns the cumulative CPU time (in seconds) used by the task
DEFINED	Returns the number of subtasks available in the current subtasking environment. The task number specified here (argument 2) must be 0 (Manager Products maintask)
DICTNAME	Returns the name of the currently open dictionary
DISP	Returns the status of the subtask and is one of the following: INIT Not yet initialized EXEC Executing WAIT Waiting
LINECNT	Returns the current output line count
LOGONID	Returns the ControlManager logon identifier
NAME	Returns the current subtask name
STATNAME	Returns the current dictionary status name
USERNAME	Returns the current dictionary user name

argument2 is mandatory and must consist of a numeric value from 0 - 99 representing the required task number. If 0 is specified then details of the Manager Products maintask are returned, if applicable.

0 is invalid for these *argument1* values:

CCOD
COMMAND
DISP
LINECNT
NAME.

A value of -1 is returned when the requested information is not available.

A value of -2 is returned when information is requested for a non-existent task.

A value of -3 is returned when a subtasking environment does not exist.

The SUBTASK function can be used by both the Manager Products maintask and all subtasks.

Examples

SUBTASK (DICTNAME, 3) Dictionary name for subtask 3

SUBTASK (STATNAME, 0) Status name for maintask

SUBTASK (LINECNT, 5) Output line count for subtask 5.

SUBTENV

This function is applicable only when a subtasking environment has been established. This function returns the task identity of the current (issuing) tasks.

➤ ➤ — SUBTENV () ————— ➤ ➤

The value returned is one of:

- 0 Task is the Manager Products maintask
- nn* Task is the Manager Products subtask number *nn*
- 3 Subtasking environment does not exist.

Example

```
SUBTENV ( )
```

TRANSLAT

```
➤➤——TRANSLAT( options )————➤➤
```

where *options* is:

```
➤——string,——[out]——,inp——[pad]——➤➤
```

The TRANSLAT function translates specified characters in string *string* into other characters or reorders them. *out* is the output translation table and *inp* is the input translation table. The output table is padded with *pad* characters or truncated to the length of the input table. The last character in the input table scanning from the left is the one that is used if there are duplicates. *string* defaults to null. *out* defaults to null. *pad* defaults to blank.

Examples

TRANSLAT('moron','&','o')	gives	'm&r&n'
TRANSLAT('sloppy','if','os')	gives	'flippy'
TRANSLAT('waste','xx','wast','.'))	gives	'xx..e'
TRANSLAT('scoundrel','','cud','+')	gives	's+o+n+rel'

TRUNCATE

```
➤➤——TRUNCATE( source-string,target-string————➤➤
➤——[F,L]——[L,R]——[I,X]——)————➤➤
```

The TRUNCATE function truncates the source-string at the point given by the target string. Truncation can be either from the left or right of the source string and the target string can be included in or omitted from the truncation.

The optional third argument specifies either the first or last occurrence of the target string as the truncation position and defaults to the first (F).

The optional fourth argument specifies either truncation on the left or right, defaulting to left (L).

The optional fifth argument specifies whether the target string is to be included in or excluded from the string when truncation occurs. The default is to include the target string (I).

Examples

TRUNCATE (abcdefg, de)	gives	'fg'
TRUNCATE (aabbccdd, c, L, R, I)	gives	'aabbcc'
TRUNCATE (xxxxyyzzz, yy, F, L, X)	gives	'yyzzz'

TYPE

There are two versions of the type function:

- The TYPE function
- The implicit type function.

ASG recommends that using the TYPE function. The implicit type function is only retained for upwards compatibility.

The two versions are described below.

The TYPE Function

➤ ➤ —TYPE(*expression*) ————— ➤ ➤

The TYPE function returns the type of the expression. The return value is one of the following:

- C for character string
- N for numeric string
- U for null string.

Examples

```
request = &P3
IF TYPE(request) EQ N THEN GOTO NUMBER

TYPE(a) || b
```

where *a* and *b* are strings.

Implicit Type Function

➤➤ `TYPE expression` ➤➤

TYPE returns the type of the expression. It returns one of the following values:

- C for character string
- N for numeric string
- U for null string.

Example

```
request = &P3
IF TYPE request EQ N THEN GOTO NUMBER
```

UPPER

➤➤ `UPPER(string` , I O) ➤➤

where *string* is a string.

The UPPER function returns the string *string* with all characters translated to uppercase.

If the second argument is omitted, a standard lower to upper case translation takes place. If the second argument is I then the input character translation set is used. If the second argument is O then the output character translation set is used.

Refer to the *ASG-ControlManager User's Guide* for more details of character translation.

Examples

```
literal #
/* 'a' = 81 in hexadecimal
/* 'A' = C1 in hexadecimal
MPR SET CHARACTER-TRANSLATION INPUT 81 C1 81 ;
upper('abCDe')      gives  ABCDE
upper(#a#)           gives  A
upper(#a#,i)         gives  a
upper(#a#,o)         gives  a
```

VALUE

➤➤—VALUE(*expression1* —————) —————➤➤
 [,*expression2*]

where:

expression1 evaluates to a variable name

expression2 evaluates to a positive integer.

The VALUE function returns the value of an element of a variable. It is particularly useful when you have dynamically allocated variable names using the INTERPRET directive.

There are two cases.

- VALUE(*expression*)
If *expression* evaluates to the name of a variable, VALUE returns the value of the first element of that variable, otherwise VALUE returns the result of evaluating *expression*.
- VALUE(*expression1*,*expression2*)
If *expression1* evaluates to the name of a variable and *expression2* evaluates to a positive integer *n*, VALUE returns the value of the *n*th element of that variable. It is invalid if *expression2* does not evaluate to a positive integer.

Example 1

```
literal #  
b = #data#  
value(b)      gives    data
```

Example 2

```
literal #  
a = #data#  
a(2) = #data1#  
b = #a#  
value(b)      gives    data  
value(b,1)    gives    data  
value(b,2)    gives    data1  
value(b,3)    gives    the null string
```

WORD

➤➤ `WORD(string, n)` ➤➤

The WORD function returns the *n*th word in the string *string*. If there are less than *n* words, the null string is returned. *n* must be a positive integer.

Examples

```
STRINGS = 'ABC DE F'
WORD(STRINGS,2)           gives    DE
STRINGS = '"ABC DE" F'
WORD(STRINGS,2)           gives    F
```

WORDINDEX

➤➤ `WORDINDEX(string, n)` ➤➤

The WORDINDEX function returns the position of the *n*th word in the string *string*. If there are less than *n* words in the string, 0 is returned. *n* must be a positive integer.

Examples

```
STRINGS = 'ABC DE F'
WORDINDEX(STRINGS,2)       gives    5
STRINGS = '"ABC DE" F'
WORDINDEX(STRINGS,2)       gives   10
```

WORDLEN

➤➤ `WORDLEN(string, n)` ➤➤

The WORDLEN function returns the length of the *n*th word in the string *string*. If there are less than *n* words in the string, 0 is returned. *n* must be a positive integer.

Examples

```
STRINGS = 'ABC DE F'
WORDLEN(STRINGS,2)         gives    2
STRINGS = '"ABC DE" F'
WORDLEN(STRINGS,2)         gives    1
```

WORDS

►►——WORDS(*string*) ——►◀

The WORDS function returns the number of words in the string *string*.

Example

```
STRINGS = 'ABC DE F'
WORDS(STRINGS)           gives  3
STRINGS = '"ABC DE" F'
WORDS(STRINGS)           gives  2
```

10

Debugging

This chapter includes these sections:

Introduction	193
SET TRACE	194
Procedures Language Trace: Selecting Procedures	194
Procedures Language Trace: Selecting Variables	195
Procedures Language Trace: Information Available	196
Manager Products Trace: Information Available	197
Examples	197
Output Media	198
SET TRACE Syntax	198
QUERY TRACE	199
QUERY TRACE Syntax	200

Introduction

This chapter describes the SET and QUERY TRACE commands.

For further information on debugging tools available in Procedures Language, see the following:

- [VLIST](#) directive (to display the contents of variables)
- [TRACE](#) directive (to display selected information while an executive routine executes)
- SET ECHO ON command (to display each command before it is executed)
- SIGNAL ON ERROR directive (to trap error conditions generated by commands)
- SIGNAL ON SYNTAX directive (to trap error conditions generated by directives).

The directives listed here are described in [Chapter 8, "Directives," on page 115](#). The SET ECHO ON command is described in the *ASG-ControlManager User's Guide*.

SET TRACE

The SET TRACE command displays information as executive routines run, to help you debug those routines; and to provide general Manager Products debug information for use in problem resolution.

The SET TRACE command enables you to obtain debug information either for Procedures Language executive routine execution or for Manager Products. The latter application is unlikely to be used on a day-to-day basis, but may be helpful as part of problem resolution, usually in conjunction with ASG personnel.

If you are debugging executive routines, you may limit tracing to particular procedures and you may trace usage of specific variables in addition to obtaining a range of debug information for each procedure. See also ["TRACE" on page 144](#).

If you are performing a general Manager Products trace, you may limit tracing to particular modules.

Procedures Language trace output may be directed to an external dataset or to the terminal. Manager Products trace may be directed to an external dataset only.

The SET TRACE command is cumulative, so that any SET TRACE commands you issue will remain in force until you switch tracing off or until you disable a particular option. To do so, enter:

```
SET TRACE OFF;
```

Procedures Language Trace: Selecting Procedures

To obtain trace information for selected procedures, enter commands of the form:

```
SET TRACE PROCEDURES name-1, name-2 ON;
```

This command switches the trace facility on for procedures *name-1* and *name-2*.

To switch tracing off, enter:

```
SET TRACE PROC name-1 OFF;
```

This command switches the trace facility off for procedure *name-1*.

You may enter the first few characters of a procedure name. In this case, all procedures whose names begin with that abbreviation will be selected for tracing.

You may select all procedures for tracing by entering an asterisk (*) in place of a procedure name.

The keyword TRACE may be abbreviated to TR, and the keyword PROCEDURES may be abbreviated to PR.

Procedures Language Trace: Selecting Variables

To trace variables selectively, enter commands of the form:

```
SET TRACE VARIABLES var-1, var-2 ON;
```

This command switches the trace facility on for variables *var-1* and *var-2*.

To switch tracing off, enter:

```
SET TRACE VAR var-1 OFF;
```

This command switches the trace facility off for procedure *var-1*.

You may enter the first few characters of a variable name. In this case, all variables whose names begin with that abbreviation will be selected for tracing.

You may select all procedures for tracing by entering an asterisk (*) in place of a procedure name.

The keyword TRACE may be abbreviated to TR, and the keyword VARIABLES may be abbreviated to VA.

The variable trace entry takes the following form:

```
V> variable(var-type,element)    procedure(proc-type,line,access)
>>data<<
```

where:

V> identifies the trace record as a variable entry

variable is the name of the user-defined variable

var-type is the type of the variable:

C	Command
G	Global
L	Local
P	Profile

element is the element number of the variable (* denotes all elements of the variable)

procedure is the name of the procedure accessing the variable. An identifier *xxx=yyyy* refers to an ASG non-procedure component.

proc-type is the procedure type:

- A Array
- C Corporate executive
- M ASG command member
- T Transient member
- U User member

line is the current line number of the procedure

access is the variable access type:

- D Drop
- R Read
- S Specify
- W Write

data is the data read from or written to the variable

Procedures Language Trace: Information Available

The options available in Procedures Language tracing are as follows:

INSTRUCTIONS	All lines are displayed as they are executed
ASSIGNS	Only assignment and DROP, PARSE and RELEASE instructions are displayed
COMMANDS	Only Manager Products commands are displayed or MPR
LOGIC	Only IF, DO, END, ITERATE, LEAVE, CALL, RETURN, EXIT, GOTO instructions, and labels are displayed
RESULTS	Displays the results of all expression evaluations for instructions being displayed by other options
CPUTIME	Displays CPU usage by command on a cumulative basis.

Using the options RESULTS or CPUTIME, with no others, also produces INSTRUCTIONS output.

When an instruction is displayed, it is prefixed by the line number of the executive routine. A plus symbol (+) following the line number indicates that the displayed line is broken into two or more lines in the original executive routine.

When a result is displayed (enclosed in single quotes) it is prefixed by >>>.

Manager Products Trace: Information Available

Manager Products trace facilities are provided to assist in problem resolution, normally with the assistance of ASG personnel. It is unlikely that you will use these facilities under other circumstances.

The options available in Manager Products tracing are as follows:

ALL	Traces modules, Manager Products messages output, virtual storage requests/releases, dynamic loads/releases, MPIN/MPOUT records
CORE	Virtual storage requests/releases
LIO	Dictionary logical block accesses
LIO-DATA	Dictionary logical block accesses and data
LOAD	Dynamic loads/releases
MESSAGES	Manager Products messages output
MODULES	Manager Products module calls
MPL	Basic procedure trace with nesting levels and return codes
ON	Minimum trace: MPIN/MPOUT records
PIO	Dictionary physical block accesses
PIO-DATA	Dictionary physical block accesses and data
RECOVERY	Recovery dataset reads/writes

Trace output will be directed to an external dataset, if allocated.

To trace modules selectively, enter a command of the form:

```
SET TRACE MODULES module-1, module-2 ON;
```

Examples

```
SET TRACE PROC £PCM,MPDY ON;          /* Trace procedures entered
SET TR PROC MYPROC ON;
SET TRACE PROCEDURES OFF;             /* Cancel procedures trace
SET TRACE VARIABLES MPDY,MYVAR ON;    /* Trace variables accessed
SET TR VAR MPCM_ARRAYGEN ON;
```

```

SET TRACE VAR OFF;           /* Cancel variables trace
SET TRACE COMMANDS ON;      /* Trace commands executed
SET TRACE LOAD ON;          /* Trace dynamic load/release

```

Output Media

By default, all trace output is directed to a ddname of MPTRACE. This name can be changed by use of the DDNAME keyword. When specified, any currently open trace dataset is closed and a new dataset opened. All previously specified trace options remain in force.

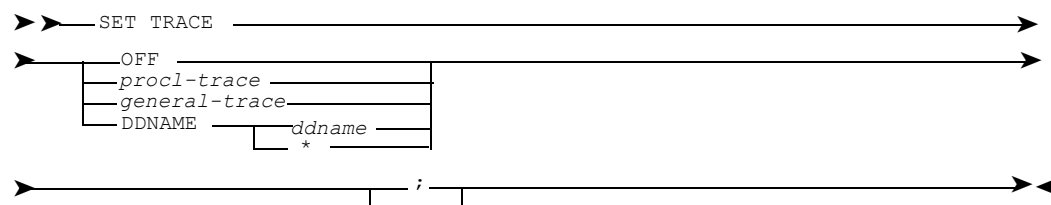
The trace dataset is opened when the first valid SET TRACE command is issued, specifying a valid trace option. Trace output is terminated and the trace dataset closed when the SET TRACE OFF command is issued. Use the QUERY TRACE command at any time to determine the trace options in effect.

To direct Procedures Language trace output to MPOUT (the terminal in online use), specify a ddname of asterisk (*). If hardcopy output is also active, trace output will only be written to the hardcopy dataset.

Manager Products trace output is never directed to MPOUT. If a ddname of MPOUT is in use and Manager Products trace output is generated, that output will be directed to a trace dataset using a standard ddname of MPTRACE.

Manager Products tracing is not available when executing under CICS. Furthermore, because an external dataset cannot be easily allocated for each Manager Products user, any Procedures Language trace output is displayed at the terminal.

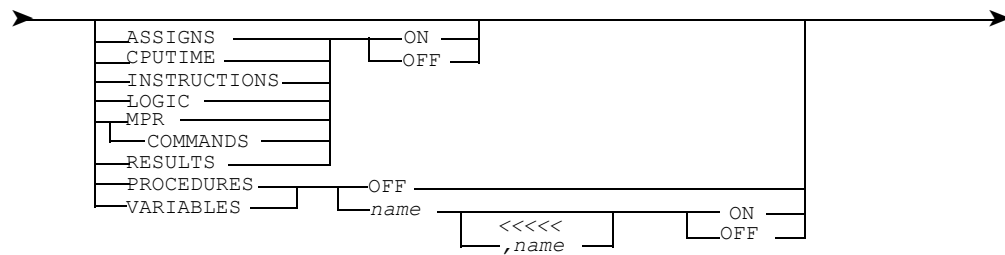
SET TRACE Syntax



where:

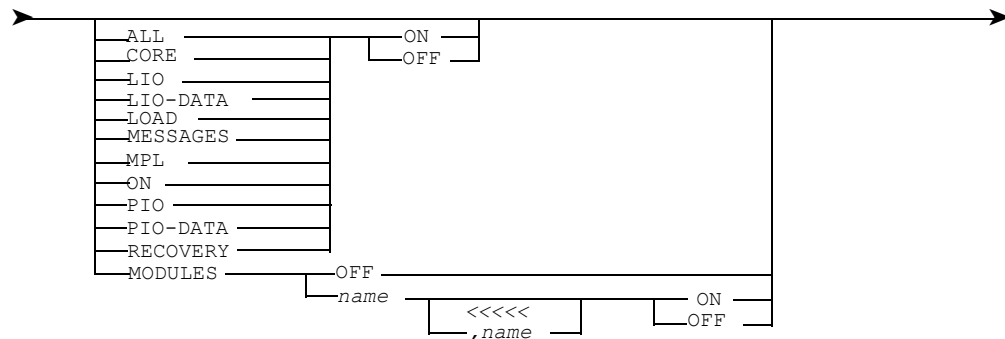
ddname is a DDNAME. It can have a maximum of 8 characters

procl-trace is:



name is the name of a procedure (maximum 10 characters) or a variable (maximum 50 characters)

general-trace is:



name is the name of a module (maximum 8 characters)

QUERY TRACE

The QUERY TRACE command displays Procedures Language and Manager Products trace settings which are currently in force. Trace facilities are invoked by the SET TRACE command. To display the current trace settings, enter the command:

QUERY TRACE;

The SET TRACE command is cumulative (that is, all trace options specified in a sequence of SET TRACE commands remain in effect until they are switched off). The QUERY TRACE command displays all such settings.

The keyword TRACE may be abbreviated to TR.

QUERY TRACE Syntax

➤➤— QUERY TRACE _____ [] ; [] _____ ➤

Glossary

Administration Dictionary

A repository controlled by the systems administrator. EXECUTIVE-ROUTINE members are held in this repository.

Array

An identifiable set of elements where each element represents an item of data.

ASG-defined Variables

These are variables that have names defined by ASG. These are the ASG-defined variables:

&G	Global Variables
&I	Installation Variables
&L	Local Variables
&P	Parameters

There are also system variables that are read-only and have values automatically assigned by Manager Products.

Command Variables

These variables exist from their declaration until the highest level executive routine terminates.

Corporate Executive Routine

A type of executive routine that can only be set up by the systems administrator. They can be used, subject to access control, by all users.

Current Line

The top line of the data area when a buffer is displayed on the screen. There is always a current line for the buffer being processed even when that buffer is not displayed on the screen (as can occur when the contents of a buffer are processed by an executive routine).

Cursor Spatial Commands

Executive routines of this type are used with the system variable &CURS. The command is usually entered by means of a PFkey. The value of &CURS is determined by the position of the cursor on the screen.

Directives

These are instructions that either control instruction sequence within an executive routine, or perform operations on data that is either internal to the executive routine or supplied by the user in the form of parameters, for example GOTO, IF and WRITEL.

Executive Commands

They are the same as primary commands, in that they may operate on external data or parameters supplied by the user. However, they may not be executed outside executive routines, as they are only applicable within the context of executive routines. For example, DACCESS is an executive command.

EXECUTIVE Members

These members reside in the MP-AID and can be used, subject to Access Control, by all users.

Executive Routine

A generic term for corporate executive routine, user executive routine, or transient executive routine.

EXECUTIVE-ROUTINE Members

These members reside in the Administration Dictionary. Once they have been constructed by the systems administrator onto the MP-AID, they become EXECUTIVE members on the MP-AID.

Function Call

An expression that returns a result. For example, the function call LENGTH('DRINK') returns a value of 5, since the string given as parameter is five characters long.

Full Evaluation

Certain expressions, for example those given following a SAY directive, are subject to Full Evaluation. Unlike Limited Evaluation arithmetic, expressions and functions are evaluated.

Global Variables

These variables exist from their declaration until the end of the Manager Products session. They may have ASG-defined names in the range &G0 to &G99 or they may have user-defined names.

Installation Variables

These variables are assigned by the systems administrator during logon to Manager Products and cannot be reassigned by the general user.

Instruction

In an executive routine an instruction specifies an action that the software is to perform. This comprises all components of an executive routine except for labels and comments. Instructions may be directives, primary commands, or executive commands.

Limited Evaluation

Certain expressions, for example those given following a WRITEL directive, are subject to Limited Evaluation. This differs from Full Evaluation in that arithmetic expressions and functions are not evaluated.

Line Commands

Executive routines of this type are normally used with parameters, and are input in the Line Command Area. The parameter values are derived from the contents of the associated data line.

Local Variables

These variables exist from their declaration until the end of the executive routine in which they were declared. They may have ASG-defined names in the range &L0 to &L99 or they may have a user-defined name.

MP-AID

The Manager Products Administrative and Information Dataset.

Parameters

Values supplied when an executive routine is invoked.

Primary Commands

These commands operate on data that is either supplied by the user or is external to that generated by the executive routine. Primary commands may be issued outside executive routines in the Command Line or within executive routines. For example, LIST is a primary command.

System Variables

These variables are automatically maintained by Manager Products and cannot be changed by any user.

Systems Administrator

A systems administrator, through the Administration Dictionary and the MP-AID, controls access to all Manager Products and repositories, and is responsible for the configuration of the Manager Products environment.

Transient Executive Routine

This type of executive routine is set up by a user as a TRANSIENT member.

TR Members

Members of this type reside in the MP-AID. They are entered directly into the MP-AID by individual users. TRANSIENTs are automatically deleted when the originating user logs off from Manager Products.

User-assigned Variables

User-defined variables can be divided into two groups: variables that cannot be assigned by the user (system variables) and variables that may be assigned by the user (all other variable types).

User-defined Functions

Functions that may be written by users in a language other than the procedures language.

User-defined Variables

These are variables that have user-defined names (up to 50 alphanumeric characters long).

User Executive Routine

This type of executive routine is set up by a user as a USER-MEMBER. It can only be accessed by the originating user or a user having the same logon identifier as the originating user.

USER-MEMBERS

Members of this type reside in the MP-AID. They are entered directly into the MP-AID by individual users, and can be accessed only by the originating user or a user having the same logon identifier as the originating user.

Variables

Locations to which names are assigned that allow data to be stored within executive routines.

Symbols

&BUFN 37
&CCOD 37
&CCOL 37
&COLO 38
&CURL 38
&CURS 38
&DATE 38
&DICT 38
&ECOD 39
&ENVM 39
&ENVT 40
&LINC 40
&LINO 40
&LOGO 40
&MODE 41
&MSLN 41
&MSLV 41
&MSNO 41
&MSTX 41
&PNUM 41
&PVAL 42
&SCOD 42
&STAT 42
&TIME 42
&TRMC 42
&TRMR 42
&USER 42

A

ABBREV function 155
ALIAS clause 86
All Occurrences routine 56
ARG function 156
ARRAYGEN executive command 64
 syntax 65
ARRAYHI function 157
ARRAYLO function 157
arrays 31
ARRAYSORT executive command 65
 syntax 66
ASG-defined variables 33

assignment
 array to array 31

B

BUILD ARRAY command 68
BUILD executive command 66
 syntax 69
BUILD KEPT-DATA command 67

C

CALL directive 116
CENTER function 158
character expressions 49
character sets 2
CLIENTI function 159
CLIENTN function 159
CLIENTU function 159
CLOSEF executive command 70
 syntax 71
command area 24
COMMAND directive 118
command variables 33
comments 9
Compound Interest routine 59
concatenation operator 46
condition information 80
CONDITION keyword 80
continuation character 8
COPIES function 160
current status information 73
cursor spatial commands 26
 example 27

D

DACCESS executive command 72
 syntax 83
DB2TYPE function 160
debugging 16, 193
Decimal Conversion routine 55
delimiters
 string 47
DEXPAND executive command 84

- syntax 91
- directives 115
 - CALL 116
 - COMMAND 118
 - DO 119
 - DROP 121
 - EXIT 121
 - general information 7
 - GLOBAL 122
 - GOTO 123
 - IF 123
 - INTERPRET 126
 - ITERATE 128
 - LEAVE 129
 - LITERAL 130
 - LOCAL 131
 - MESSAGE 131
 - MPR 133
 - MPRE 134
 - MPXX 134
 - NOP 135
 - PARSE 135
 - PARSEOPTION 137
 - PROFILE 137
 - RELEASE 138
 - RETAIN 140
 - RETURN 140
 - SAY 141
 - SET 141
 - SIGNAL 142
 - STACK 143
 - TRACE 144
 - TRANSFER 145
 - VLIST 146
 - WRITEF 149
 - WRITEL 151
- DIVCAPT function 160
- DIVOBJN function 161
- DIVOBJT function 161
- DO directive 119
- DRELEASE executive command 91
 - syntax 93
- DRETRIEVE executive command 94
 - syntax 105
- DROP directive 121
- E**
- EDDATE function 161
- EDTIME function 163
- efficiency 20
- evaluation
 - full 47
 - limited 48
- examples
 - cursor spatial commands 27
 - executive routines 51
 - line commands 25
 - switching output 34
- execution control 10
- executive commands
 - ARRAYGEN 64
 - ARRAYSORT 65
 - BUILD 66
 - CLOSEF 70
 - DACCESS 72
 - DEXPAND 84
 - DRELEASE 91
 - DRETRIEVE 94
 - RELINQUISH 106
 - RESERVE 107
 - SENDF 108
 - SREAD 113
- executive routines
 - corporate 21
 - examples 51
 - running
 - from Command Area 24
 - from Line Command area 24
 - transient 24
 - user 24
- EXECUTIVE-ROUTINE members 23
- EXIT directive 121
- expanding member for particular language 85
- expressions 45
 - character 49
 - numeric 49
- EXTRACT function 163
- F**
- FASTQUIT routine 54
- FDO function 166
- functions
 - ABBREV 155
 - ARG 156
 - ARRAYHI 157
 - ARRAYLO 157
 - CENTER 158
 - CLIENTI 159
 - CLIENTN 159
 - CLIENTU 159
 - COPIES 160
 - DB2TYPE 160
 - DIVCAPT 160
 - DIVOBJN 161
 - DIVOBJT 161
 - EDDATE 161
 - EDTIME 163

-
- EXTRACT 163
 - FDO 166
 - general information 12
 - GETSVRM 167
 - GETTOKEN 168
 - GETUDSN 168
 - HEX 169
 - INSERT 169
 - LASTPOS 170
 - LEFT 170
 - LENGTH 170
 - LOWER 171
 - MAX 172
 - MEMTYPE 172
 - MIN 173
 - MPRAID 173
 - MPRCMPW 174
 - MPRDDPW 174
 - MPRSU 174
 - MPRUCLS 174
 - MPRUDSN 175
 - NDATE 175
 - NTIME 176
 - OVERLAY 177
 - PARSABLE 178
 - POS 178
 - PTIME 178
 - REDUCE 179
 - REPSTR 179
 - REVERSE 180
 - RIGHT 180
 - ROOT 180
 - SEARCH 181
 - SERVERN 182
 - STIME 182
 - STRIP 182
 - SUBSTRING 184
 - SUBTASK 185
 - SUBTENV 186
 - TRANSLAT 187
 - TRUNCATE 187
 - TYPE 188
 - UPPER 189
 - user-defined 13
 - VALUE 190
 - WORD 191
 - WORDINDX 191
 - WORDLEN 191
 - WORDS 192
- G**
- GETSVRM function 167
 - GETTOKEN function 168
 - GETUDSN function 168
- H**
- GLOBAL directive 122
 - global variables 34
 - glossary 201
 - GOTO directive 123
- I**
- IF directive 123
 - INSERT function 169
 - installation variables 36
 - INTERPRET directive 126
 - ISPF command examples 60
 - ITERATE directive 128
- L**
- labels 7
 - language
 - expanding for a particular 85
 - LASTPOS function 170
 - LEAVE directive 129
 - LEFT function 170
 - LENGTH function 170
 - line commands 24
 - example 25
 - LITERAL directive 130
 - LOCAL directive 131
 - local variables 35
 - LOWER function 171
- M**
- MAX function 172
 - MEMTYPE function 172
 - MESSAGE directive 131
 - MIN function 173
 - MP-AID Copy routine 51
 - MP-AID List, Condensed example 52
 - MPR directive 133
 - MPRAID function 173
 - MPRCMPW function 174
 - MPRDDPW function 174
 - MPRE directive 134
 - MPRSU function 174
 - MPRUCLS function 174
 - MPRUDSN function 175
 - MPXX directive 134
- N**
- name conflicts 27
 - NDATE function 175
 - NOP directive 135

NTIME function 176
numeric expressions 49

O

operators 45
OVERLAY function 177
Overlay routine 57

P

PARSABLE function 178
PARSE directive 135
PARSEOPTION directive 137
parsing 18
PF Key Settings routine 58
POS function 178
PROFILE directive 137
profile variables 34
PTIME function 178

Q

QUERY TRACE command 199
 syntax 200
Quick Sign On routine 54

R

readability 20
REDUCE function 179
RELEASE directive 138
RELINQUISH executive command 106
 syntax 107
REPSTR function 179
RESERVE executive command 107
 syntax 108
RETAIN directive 140
return codes 42
RETURN directive 140
REVERSE function 180
RIGHT function 180
ROOT function 180

S

SAY directive 141
SEARCH function 181
security information 73
SENDF executive command 108
 syntax 112
SERVERN function 182
SET directive 141
SET TRACE command 194
 syntax 198
SHOW MEMBER-TYPE command 94
SIGNAL directive 142
SREAD executive command 113

 syntax 114
STACK directive 143
STIME function 182
STRIP function 182
SUBSTRING function 184
SUBTASK function 185
SUBTENV functions 186
SUPPRESS clause 77
system variables
 &DICT 38
 &ENVM 39
system variables 37
 &BUFN 37
 &CCOD 37
 &CCOL 37
 &COLO 38
 &CURL 38
 &CURS 38
 &DATE 38
 &ECOD 39
 &ENAM 39
 &ENVT 40
 &LINC 40
 &LINO 40
 &LOGO 40
 &MODE 41
 &MSLN 41
 &MSLV 41
 &MSNO 41
 &MSTX 41
 &PNUM 41
 &PVAL 42
 &SCOD 42
 &STAT 42
 &TIME 42
 &TRMC 42
 &TRMR 42
 &USER 42

T

TRACE directive 144
TRANSFER directive 145
TRANSLAT function 187
TRUNCATE function 187
TYPE function 188

U

UPPER function 189
user-defined commands 2
user-defined variables 31

V

VALUE function 190

- variables
 - array 31
 - ASG-defined 33
 - command 33
 - global 34
 - installation 36
 - local 35
 - naming rules 31
 - parameter 36
 - profile 34
 - releasing rules 92
 - summary 30
 - system 37
 - system-assigned 33
 - user-assigned 33
 - user-defined 31
- VLIST directive 146

W

- WITH STATUS-DETAILS clause 76
- WORD function 191
- WORDINDX function 191
- WORDLEN function 191
- WORDS function 192
- WRITEF directive 149
- WRITEL directive 151

